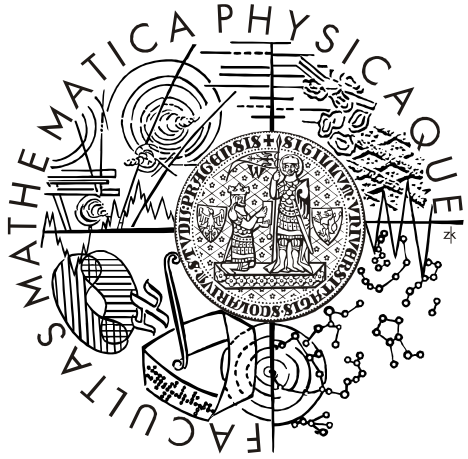
Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS



Bc. Tomáš PLCH

Action selection for an animat
Mechanismus výběru akcí pro animata

Department of Software and Computer Science Education
Supervisor: Mgr. Cyril Brom, Ph.D.
Study program: Computer Science, Software Systems

Acknowledgment

I would like to thank all my friends and family for motivation, support and inspiration. I also would like to thank my supervisor, for all his insight and patience.

I declare that I have written this thesis by myself and that I have used only the cited resources. I agree with making this thesis public.

In Prague 16. 4. 2009 Tomáš PLCH

Contents

Název práce: Mechanismus výběru akcí pro animata

Autor: Bc. Tomáš PLCH

Katedra (ústav): Kabinet software a výuky informatiky

Vedoucí diplomové práce: Mgr. Cyril Brom, Ph.D.

e-mail vedoucího: Cyril.Brom@mff.cuni.cz

Abstrakt: V tejto práci je skúmaná metóda reaktívneho plánovania, ktorá predstavuje populárnu voľbu pre riadenie správania umelých bytostí. Výhodou tohto prístupu sú okrem iného jednoduchosť návrhu, rýchlosť a uveriteľné chovanie agentov v dynamickom a zložitom prostredí. Vďaka jednoduchosti celého konceptu je problematické modelovať zložitejšie formy chovania spôsobom, aby výsledné chovanie bolo v určitých prípadoch uveriteľné a považované za inteligentné. Táto práca je zameraná na rozvinutie konceptu reaktívneho plánovania spôsobom, aby bolo možné pozorované nedostatky v dostatočnej miere kompenzovať pri zachovaní základných princípov modelu. Práca predstavuje sadu návrhov pre zmenu formy akou je chápané reaktívne plánovanie pridaním fázového modelu ako alternatívy pre základnú štruktúru reaktívneho plánovania – reaktívneho plánu. V práci sú rozobraté i praktické problémy výberu akcie pre animata. Súčasťou práce je aj jednoduchý prototyp, ktorý na krátkych scenároch prezentuje niektoré z riešených obmedzení reaktívneho plánovania.

Klíčová slova: reaktívne plánovanie, fázy, animat, umelá bytosť, uveriteľné správanie, umelá inteligencia, agent

Title: Action selection for an animat**.**

Author: Bc. Tomáš PLCH

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Cyril Brom, Ph.D

Supervisor's e-mail address: Cyril.Brom@mff.cuni.cz

Abstract: In this thesis we study the method of reactive planning, which is very popular for modeling behavior of virtual beings. Advantage of this approach is in the simplicity of design, its speed and believable behavior of agents in dynamic and complex environments. Thanks to the simplicity of the concept, it is rather difficult to model complex forms of behavior in a way, that the resulting behavior is believable and perceived intelligent. This thesis is focused on the expanding the concept of reactive planning in such a fashion, to compensate for observed limitations, preserving the basic concepts. The thesis contains proposals for various improvements to the basic structure of reactive plans – the reactive plan. In the presented work, we analyze practical problem of action selection used in reactive planning. Part of the thesis is a simple prototype, which is used to present some of the tackled limitations of reactive planning.

Keywords: reactive planning, phases, animat, artificial being, believable behavior, artificial intelligence, agent.

1 Introduction

Virtual agents are becoming very popular in the both academic and industrial domain, used in computer simulations and various applications. Thanks to the growing processing power of commonly available computer equipment, they also became appealing for general public, most notably in computer games [1], interactive worlds or education in the field of artificial intelligence [2].

The virtual worlds we look upon today have gained on the one hand in visual finesse, on the other hand, increased in complexity. These better-looking worlds create a demand for more realistic environments inhabited by interactive, intelligent and believable autonomous agents. A popular example is the Mass Multiplayer Online Role Playing Game (MMORPG) World of Warcraft [9] seen in [Picture 1].

Applications of virtual agents range from movie industry, computer games, academic studies [3], trough virtual storytelling [4], military simulations [5][6] and civilian applications [13] to behavioral modeling [7]. An overview of various applications can be found in [7].

Picture 1 - World of Warcraft in-game screenshot

A virtual agent can be viewed as an autonomous system that processes the percepts from the environment it inhabits and takes actions satisfying a goal or following a task [12], given by the designer or acquired by the agent himself - from other agents, from the environment or from a perception of the world. The agent can be put into a dynamic and unpredictable environment, thus making it more difficult to achieve its goals in timely and believable fashion. A virtual agent might be equipped with a virtual body, allowing interaction with the environment in a more realistic way.

Most researchers now agree, that intelligence is a manifestation of behavior [16]. Systems that use behavior to model intelligence are known as *Behavior-based Systems* (BBS) [17], opposite to the *Knowledge-base Systems*. The idea of BBS is based in ethology [18] – a branch of biology that studies animal behavior patterns.

The main concern in virtual agents research and application is to mimic intelligence[1] by behaving in a believable fashion, focusing on visualization and action selection. The aspect of proper visualization gives the agent and virtual world a more believable look, where selecting adequate actions in a timely fashion gives the feel of intelligence. When adopting the idea of intelligence manifested trough behavior, an agent can be perceived intelligent when doing it “right” in the eyes of the observer (in most cases a human). An agent that

[1] Respective to the role of the agent in the simulation

bumps ten times into a wall or takes minutes to decide its next action can be considered inadequate for most applications, when observed in real-time (games, online simulations etc.).

The action selection is also known as the *action selection problem* (ASP) – which action will be selected for execution from the domain of all for the agent available actions. The *action selection mechanism* (ASM) is the method by which the next action will be chosen [19].

Various ASM are employed, to name only a few – Internal Behavior Network (IBeNet) [19], the IDA and LIDA architectures [20] or Reactive planning [21].

1.1 Motivation

The virtual worlds gain with every generation on complexity and the agents in those worlds are required to show believable behaviors without increasing the computational complexity of their brains [32]. When limited processing power is divided upon multiple agents and the world mechanics, a single agent's brain[1] cannot hope for many CPU cycles for its personal use. Therefore, it has to stay as simple as possible but provide a sophisticated vehicle for the world designer to create interesting and believable behavior, providing an illusion of intelligence.

To achieve that, the action selection and its results are the most important part of the virtual agents artificial mind. The techniques employed to create the agent's ASM include hierarchical finite state machines [35][25], reactive planning [22][26], scripting or use of genetic algorithms [27]. Employing more complex techniques, like proper reasoning [28], machine learning can lead to more computationally complex minds and as a result, less responsive, slow agents. The aspect of easy design and debugging is important - to allow specialists (or students), who are not software developers, to contribute to the creation of an agents mind [23][2].

From all this – agile, believable, complex, entertaining, and easy-to-design entities in a dynamic and complex environment – we chose the believability of behaviors for agents, to be the most motivating task to tackle. The demand on computational less complex mechanics lead us to employ the Reactive Planning techniques presented by J. Bryson in [21], later discussed in [24] by C. Brom, presenting various limitations that make it hard or unable to develop certain traits or behaviors natural to living beings. Our main motivation is to provide compensations for those limitations, to be able to bring the concept of reactive planning further, allowing us to simulate even more complex virtual beings, providing a better illusion of them being "smart".

1.2 Thesis structure

This thesis is divided into chapters.

The 1st Chapter is devoted to the introduction into this thesis. The 2nd Chapter lists an overview of the thesis goals to provide a picture of the problems at hand. The 3rd Chapter is devoted to explain the basic theory behind the *reactive planning* and *action selection* concepts.

The fourth chapter is presenting the problems of selecting proper actions. Chapter 5 is committed to present the limits of reactive planning, based upon [24]. In Chapter 6, we provide our improvements to the basic structures of reactive planning and extensions to ASM. The concluding Chapter 7 is devoted to present an action-based approach to solving a specific problem – *intentions*. Chapter 8 describes our prototype.

[1] The Action Selection Mechanism being a part of it, choosing the agents actions – creating the behavior.

The 9th chapter is dedicated to summarize our findings, concepts and presenting scenarios used in the prototype. Our overall conclusion is drawn in the 10th Chapter. Chapter 11 is concerned with future work related to the topic of this thesis.

The Appendix A contains instructions on how to install the provided prototype. Appendix B is a listing of related work and Appendix C contains information about the appended CD-ROM.

1.3 Terms and notation

The basic term to explain is the *agent*, also referenced as the *animat* or *artificial entity*. We also use the term *actor*, because the presentation of our improvements is without the direct influence of the observer. We interchange these terms, but they represent the same thing.

Actor/Animat/Agent is an autonomous entity that is designed to carry a set of objectives given by a designer or acquired during existence. It exists in an unpredictable and dynamic environment performing actions based upon percepts from the environment and its internal *action selection mechanism* (ASM). Its behavior should be believable considering its apparent role[1].

It is important do distinguish the real examples in our scenarios and their counterparts in the simulation. Objects and entities that are referenced in the real world are denoted prefixing '*r-'* (r-*dog, r-bucket*) and their virtual counterparts are denoted with a prefixing '*v-'* (*v-dog, v-bucket*). Sometimes, when the *real* and *virtual* can be distinguished based on the context, we tend to leave the prefixes out.

activity – a set of actions performed by the *r-agent* (human) in the *r-world*

actions – process in the world (real and virtual alike), that might modify objects or the world, performed by an agent. We can distinguish *external* - performed to modify the world and *internal* – performed to modify the agents mind[2].

plan – a set of actions that follows an overall concept – a goal. It provides the agent with actions it can perform.

v-world – the virtual environment, where the v-actors reside and act, considered complex and dynamic. V-actors can perceive and influence the v-world by their v-actions.

v-object – objects in the *v-world,* can be collected and used.

[1] Virtual dogs should behave like dogs, virtual humans like humans.
[2] Actions can influence other parts of the mind e.g. memory, mood, emotions etc.

2 Thesis goals

The main goal of this thesis is to improve the idea of *reactive planning* to compensate for the problems and limits observed in [24] and [15]. Our goal is to extend the techniques and structures of reactive planning to provide better and more believable behavior simulation. We also propose additional mechanics[1] that could contribute to the better performance of the *reactive plan* concept in an artificial mind.

We implemented a simple prototype to demonstrate some of our techniques on simple scenarios.

2.1 Reactive planning overview

Reactive planning denotes a set of techniques for *action selection mechanism* (ASM) used to handle *autonomous agents,* residing in unpredictable and dynamic (virtual) environment. Also known by the term *dynamic planning,* reactive planning exploits the idea of *reactive plans,* which consist of *condition-action* rules.

A condition-action, also known as *if-then* rule, has a simple form

if *condition* **then** *action*

Representing a simple concept – ***if*** *condition* holds ***then*** perform action. Action selection mechanism (ASM) evaluates these rules at every given opportunity (periodically) choosing next one to be performed.

In a *reactive plan*, the condition-action rules are ordered by their *priority*. A *rule* is chosen for execution based upon its priority and a *holding condition,* maximizing the *priority.* The actions can be viewed as *internal*[2] or *external*[3].

A rule that is chosen for execution is called *active* or *executing.* Rules with holding conditions are called *preactive* and all others are *inactive.* Rules, which were active but are surpassed by higher priority rules, are called *suspended* or *switched*.

Action selection mechanism and reactive plans in reactive planning are described in [Chapter 3] in more detail.

2.2 Problems and limits to overcome

When employing reactive planning in its simple (flat) or hierarchy structure, multiple problems emerge, rendering the resulting agents behavior appear less intelligent to a human observer. Intelligence, manifested trough behavior [16], has to mimic a human conception of the behavior perceived. It is necessary to identify and compensate for the limitations that can hinder the illusion of believable behavior [24].

These limitations can be split up into categories:

- *intentions*
- *planning and deliberation*
- *transitional behavior*

Every category embraces various limitations. Our goal is to overcome them by enriching reactive plan structure [Chapter 6] by introducing *phases* and *auxiliary structures,* improving information flow inside reactive plans and exporting information to the enveloping structures and handling algorithms. In [Chapter 7] we introduce the

[1] Extended ASM and Extended Actions

[2] Affecting internal states, structures and mechanisms of the agents mind – memory, mood, focus...

[3] Pick up or use objects, move the agent, perform actions or change the agents external properties (smile, move hands, perform gestures)

Intention-Goal-Plan map (IGPmap) and the *Intention-add Action (IaA)* – which provide an approach to overcome *intention* related issues.

2.2.1 Intentions

Based on the *Belief-Desire-Intention* [12] model, the *intention* is a deliberative state, something the agent wants to achieve. An intention can be achieved by satisfying a set of *goals*, which are satisfied by *reactive plans*. Put together

$$intention \rightarrow goals \rightarrow plans$$

Intentions associate with goals, which are accomplished (satisfied) by plans.

Reactive planning provides only static structures with no capability to add, remove or manage intentions (with associated goals and plans), rendering the agent less receptive to newly introduced situations, able to solve only those who are hard-wired into his brain.

A reactive agent is also not capable of choosing alternatives for satisfying his goals, because plans in reactive planning are mapped to single goals.

2.2.2 Planning & deliberation

Reactive planning eludes planning and deliberation to provide responsive behavior. Nevertheless, humans tend to look for a degree of planning when perceiving intelligence. It gives a feel of agent having awareness not only of surroundings, but also of his internal states and a tendency to reach a specific state in the future. Illusion of planning gives a feel that the agent actually knows what is happening and what will happen next, able to steer it according to his intentions (manifested desires).

The main disadvantage of reactive plans is the lack of coordinated preparation for plans in execution and plans that will be executed shortly. The problem can expand either into the deeper level of the hierarchy, or to plans following each other in execution. Easily described as a “lack of collecting objects along the way”.

The structure of reactive plans is not suited to perform optimized searches and retrieval of items except for the order represented by the if-then rules.

Searching objects can get pretty tricky, especially in reactive plans, when sharp timeouts are introduced to limit the time spend searching. Introducing timeouts into conditions is a rational step, but can lead to absurd behavior – expired timeouts can break off the search having the object just few steps away.

Optimizing behavior execution by creating a chain of activities that have common objects and location is natural to humans. Reactive planning executes only the rules or plans with the highest priority, not concerned about the possible link between them. Therefore, optimizing a chain of plans, sub-plans or actions is not supported and represents a limitation.

2.2.3 Transitional behavior

Transitional behavior is a short sequence that is performed between two plans – the *suspender* and the *suspended*. The suspended plan is *discontinued* and possibly put into a consistent state to be resumed later. The suspender plan replaces the suspended and is executed. It also can be called “a switch” of plans.

Transitional behavior is common, but it can be hard or impossible to model this using reactive plans. Reactive planning only executes plans that have the highest priority not caring about those with the lower priority, even when executed in the previous step.

2.2.4 Cleanup behavior

The *cleanup behavior* is a form of transitional behavior, which humans tend to perform after an activity has finished – putting used items back from where they were taken.

Cleanup behavior can be modeled using reactive plans, adding the cleanup before the explicit ending of the plan. This approach might not work in every situation - e.g., external induced fails or aborts. Also the precise ordering of dictated by their priority can lead to ineffective order of cleanup. It is similar to optimized searches addressed earlier. When plans share objects, a preliminary cleanup of those shared objects by the finishing plan can lead to unnecessary pickups by the active plan providing a less believable image of the agents behavior[1].

2.2.5 Behavior aftermath

Humans tend to evaluate the result of their activities – simply put – we are happy about success and frustrated of failure. This can be seen as *behavior aftermath,* where certain actions (internal) are executed based upon the result of the plan.

There is no concept of *aftermath* in reactive planning; rendering reactive plans and ASM unaware of the results produced (success or fail).

2.2.6 ASM related problems

Our goal in this thesis includes identifying and compensating for problems originating in the ASM algorithm that can evoke the feeling of less believable behavior or hinder smooth execution.

- *interrupting*
- *condition fault*
- *delaying rule activation*
- *failure & success*
- *random rule selection*

The basic idea of ASM is to find a rule with highest *priority.* When two or more rules with equal highest priorities have holding *conditions,* one is chosen at random.

We tend to these problems in more detail in [Chapter 4].

2.2.6.1 Interrupting – behavior consistency

Humans often abstract and view behavior as something more complex then just simple atomic actions, perceiving sequences of actions that have to follow distinct order and keep consistency. Interrupting consistent behavior patterns can be viewed as a disorder.

We need to take constrains of the virtual world into account. Some action sequences are demanded to be uninterrupted and consistent, to keep the visualization or environment of the virtual world intact.

The *interrupting* can pose a serious problem for ASM. When demanding longer sequences to be consistent and the environment is dynamic enough to provide input for ASM to interrupt those sequences, without modification, behavior consistency cannot be (easily) guaranteed.

[1] The agent puts the object down and in the next instant, picks it up.

2.2.6.2 Condition fault

When single atomic actions are used, this won't occur. On the other hand, when longer sequences (2 and more actions) are employed, a *condition fault* can occur during the execution of a sequence. A condition fault is a situation, where the *condition* of the rule ceases to hold in mid-execution of an associated action sequence, either by external or internal (by the sequence itself) cause. Plain ASM won't continue executing the action, not considering it in the search of a candidate for the active rule.

This represents a significant problem, when action sequences disrupt their own *condition,* rendering it *false,* ceasing their execution before they finish[1].

2.2.6.3 Delayed rule activation

Delayed rules are a problem tightly bound to the interrupting problem (2.2.6.1). Rules of higher priority that have a holding condition should be chosen by the ASM, but they can't, because of an executing rule with consistent behavior (2.2.6.1). It can be deemed necessary to put such rule to execution anyway, possibly later. Based on this assumption, the rule has to be delayed and considered by the ASM later, even when the condition of the delayed rule ceased holding (it held in the past).

Compensating this problem can lead to more believable behavior, keeping the rule execution in a more consistent appearance – rules will execute, with a short, hardly noticeable delay.

2.2.6.4 Fail & Success

The first problem with a fail/success of plans is how to propagate them. When a plan is explicitly failed or succeeded, it should be distributed to its active parts residing downwards in the hierarchy (beneath the "source" of he fail/success). The ASM ignores the fail/success continuing on searching the next rule for execution, only resetting the current failed/successful branch of the hierarchy.

The other issue is on repeating faulty behavior – a search for a active rule may always end in a rule which ends in a failure[2]. ASM doesn't recognize that a faulty event is reoccurring, providing a repeating faulty behavior, rendering the overall impression less believable.

2.2.6.5 Random rule selection

The random choice mechanism for rules with equal priorities can be insufficient. In some cases, it can be demanded to prefer some rules based on their significance.

To illustrate this – dogs tend to run around and bark. They don't do these actions on an equal basis. They prefer to run around a lot, occasionally barking. When unbiased random rule selection is employed, this renders a minor but recognizable drawback.

2.3 Prototype

We created a prototype simulation to present some of our solutions on simple scenarios. The prototype doesn't cover all the improvements we proposed in this thesis, because some of them require extensive simulations and too complex situations to manifest properly. It is necessary to note, that some concepts can be fully exploited only in a conjunction with a fully working v-world engine.

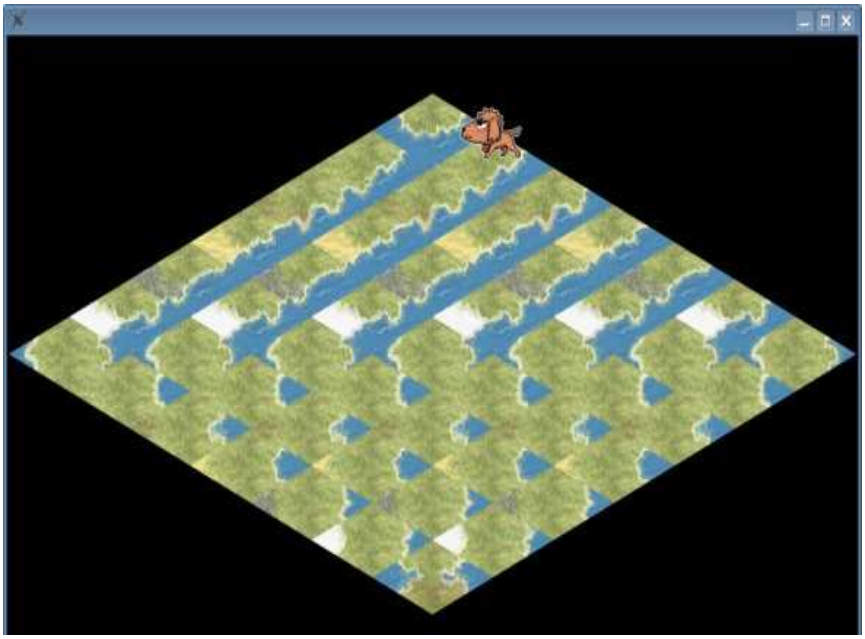
[1] They will never finish, always corrupting the condition
[2] An agent stands in front of a locked door and tries to open it.

2.4 Conclusion

In this chapter, we introduced a preview of the problems that represent the limits we want to compensate. We consider reactive planning to be a strong instrument able to provide believable behavior for agents, but the uncontrolled way of execution and overall simplicity can lead to the observed constrains.

Our main goal is to introduce techniques and modifications, to allow better control and means to analyze the plans to be able to optimize cooperative execution, ordering and runtime modification and adaptation to the emerging situations. By compensating for lack of control and information flow from plans to action selection and other plans, we attain to compensate for the emerging limitations. Keeping the main idea of the overall concept unchanged, keeping the basic properties of speed and responsiveness in mind.

The later chapters are devoted to further explain the problems of ASM and limits of reactive planning and present our improvements to the given concepts, to circumvent or compensate these issues, providing a more believable behavior[1].



Picture 2 Prototype screenshot

[1] In respect to human perception.

3 Action selection – How reactive plans work?

Humans consider the behavior of real world entities as implicitly believable[1]. Therefore, humans require virtual counterparts of living entities to behave similar to be believable and appear intelligent.

Action selection is an important part in the *life* of a virtual agent - the action it chooses, are the manifestation in the virtual world and are observed and judged as (maybe) intelligent behavior. It is important that the chosen actions are appropriate to the actual situation and satisfy the agent's goals, either in short or long term horizon. The goals is either given by the designer or acquired during existence – emerging from a situation, encounter or a state of mind. An important factor is the speed of the action selection, where a timely fashion is considered more believable.

In our model world, every agent can perform atomic actions that take up one time-cycle. The actions can influence the world in various ways – modify, move, create objects etc. It is up to the designer to arrange these actions so the resulting behavior is believable.

We consider the reactive planning action selection mechanism to be a good candidate for a suitable *action selection mechanism* (ASM) of an artificial mind. Of course with adjustments to further improve the believability of the final behavior.

3.1 Why not classic planning?

Lets consider a simple scenario – "Going to buy some groceries" - and comparing *r-world* and *v-world*. The scenario consists of two actors – Bob and Tom, where Bob has the unpleasant task to go buy groceries to a nearby shop and he meets Tom on the way there. Toms task is to speak to Bob when they meet.

Considering the r-world situation first. When *r-Bob* meets *r-Tom*, they both stop for a short chat. After that, r-Bob continues to the shop and finishes his task of buying groceries. There is a small possibility a car hits r-Tom. In conclusion, r-Bob and r-Tom don't meet, and r-Bob continues on his journey to the shop unhindered.

When in v-world, and v-Bob uses the classical planning, first he plans his whole route to the shop, without knowing, that v-Tom might intercept him on the way. He walks the whole way to the shop, ignoring v-Tom when they meet. After finishing shopping, v-Bob could create a plan to talk to v-Tom, but v-Tom could be gone already, fulfilling a different goal of his own. It can be argued, that v-Tom could interrupt v-Bobs execution of the plan, but in more complex environments, where interrupts could lead to recompiling the plan too often, rendering the agent unresponsive and less agile in completing its goals.

This simple scenario shows how behaviors based on classical planning in dynamic environments can be less effective and less believable. The main disadvantages of classical planning are the time consuming calculations of the in-forward created plan and agents low rate of responsibility to sudden changes - a new plan has to be computed all over again. The action selection done with classical planning creates an ordered action set that leads the v-actor from its starting point to the grocery shop, not considering the events, or conditions that could happen along the way. Too complex and dynamic environments can lead to large planning domains for the classical planning to cope with.

There is also the issue of internal drives of agents, which can emerge at any time, like hunger, sickness or internal states of mind, like emotions, moods or psychosis.

[1] Living entities, like dogs and humans are considered impliclitly intelligent due to their biological nature, providing the model for believable behavior.

From a designer's point of view, these events occur in an asynchronous way, thus adding more dynamic to the inner and outer virtual world.

The "Going to buy some groceries" scenario is a good example, how Reactive planning can be employed with better (more believable) results, then classical planning. The actions selected by reactive planning's action selection are computed in every cycle based on the actual world context. The planning domain is limited to the plans and their structure complexity, and therefore can be better optimized and more responsive and adaptive. To illustrate this on the given example, v-Bob considers what to do after every step he makes. His selected actions are walking towards the shop, but when he meets v-Tom, the actions related to the meeting have higher priority, and therefore are chosen and v-Bob stops walking and starts talking to v-Tom.

This behavior is more believable then the one portrayed in the classical planning version of the scenario. To further advance the picture, the conversation actions can be conditioned by an internal state of v-Bob, so he rushes by v-Tom ignoring him, when he is angry or is in a hurry.

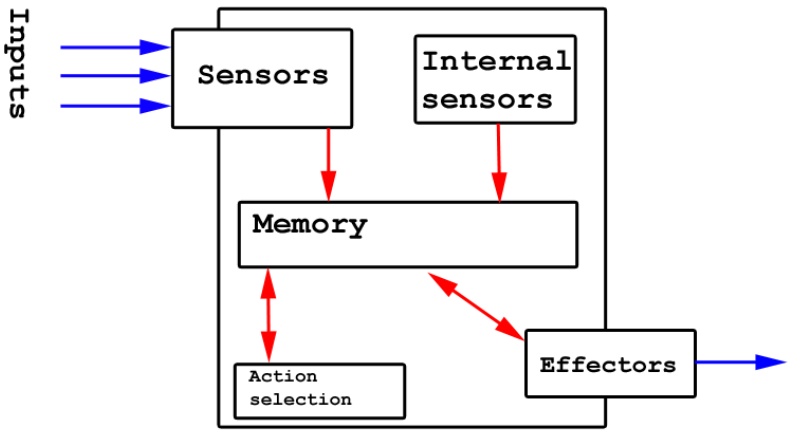
3.2 Reactive planning

Reactive planning can be considered a misleading term. In general, this approach is not about planning in its common acknowledged sense, viewed by the classical planning (discussed above). In classical planning plans are created to perform a task or achieve a goal by available actions in forward, where the ordering is an essential key. The precomputed plan, if interrupted, has to be (in most cases) recomputed, to achieve consistency.

Reactive planning differs from classical planning in many aspects, notably, the timely fashion it can cope with dynamic and unpredictable environments and the absence of forward planning. It is common, that the preconstructed plans are given by a designer and make up the action selection mechanism of the artificial mind - the agent follows a set of given "plans", in contrary to creating the plan on its own. The reactive plan is not an action sequence, like in classical planning, it is a condition-execution based decision structure consisting of rules and their boolean conditions, that represent the "context of the world" when the action is to be considered for execution.

The basic idea of reactive planning is to choose an action based upon the context that is derived from the environment, where an autonomous agent resides and the internal state of its mind and memory. The memory can be used as a middleware between the sensors collecting percepts from the environment and the other components of the mind, like action selection, allowing more general approach to development and more independent components [Picture 3]. A sophisticated memory-mind model can be seen in [8].

We will present our understanding of the Simple Reactive Planning (SRP) concept proposed by J.Bryson, extended into Hierarchical Reactive Planning (HRP)[14].



Picture 3: Memory middleware

3.3 Simple Reactive Planning (SRP)

Simple Reactive Planning (SRP) can be considered the flat version of reactive planning embracing only a one level architecture of *condition-execution rules.*

The basic building block of SRP is a *reactive plan*. In [14] J. Bryson refers to it as *basic reactive plan* (BRP). Formally a reactive plan is a set of *if-then* rules meaning, "***If** (boolean condition) is true **then** perform action"*. This condition-based execution is the basic idea of the action selection mechanism (ASM). Conditions are evaluated in (almost) every world cycle, performing the associated actions, only one action per evaluation cycle per agent. Thanks to this concept, ASM is able to providing the agility and responsiveness needed in an unpredictable environment.

An *if-then rule* can be formalized as a *triplet - {prio, cond, exec}.*

- *prio* denotes the priority of the *rule* in the containing reactive plan
- *cond* is the boolean-based condition
- *exec* is the action (not being an atomic action in this context).

We call the *if-then rule* simply a *rule*, specifying if the context indicates otherwise. The reactive plan can be abbreviated into *plan* when there is no doubt that we are talking about reactive plans.

The set of rules in a reactive plan is ordered by priority. The basic idea of the ASM is to find a rule with the highest priority and a condition (*cond*) evaluated *true*. In the case of more rules of the same priority and a holding condition, one of them is chosen by random. The found rule is called *active rule* and its action is executed in the current step. All rules with holding conditions are *preactive.* All other rules are called *inactive.* Rules formerly executed but surpassed are called *suspended.*

The importance of a *rule* is specified by its priority (*prio*), in most cases represented with and integer. Rules with higher priority are considered to be more important and are executed on behalf of the lower priority rules. As discussed in [15], the priority could be a *function* of time, internal and external context, thus making the *cond* parameter redundant. The condition would be „wired" into the *function*, manipulating the priority, promoting the best candidate based only on priority.

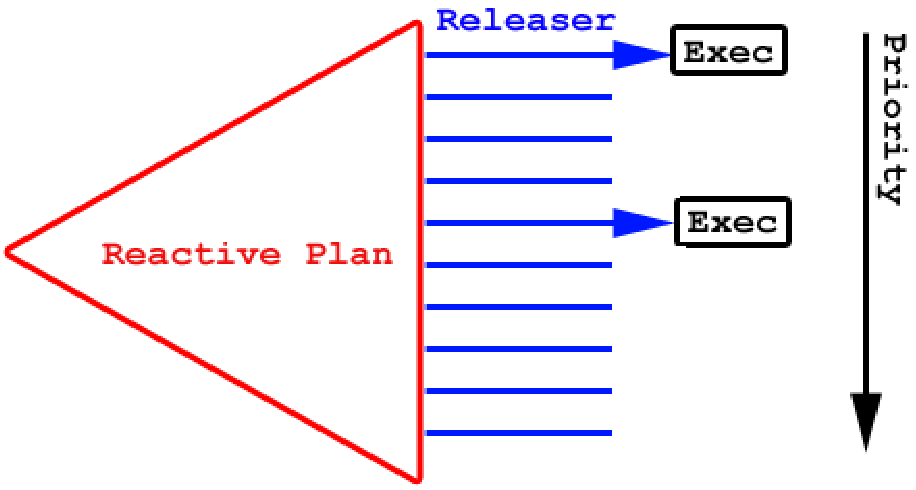
This approach has a major drawback, where all the functions have to be evaluated to receive all priorities, to determine the active rule. In plain SRP model, the ASM only checks the rules in a descending order, checking for the first holding condition.

The condition (*cond)* can be virtually anything that results in a boolean expression. It can be implemented as a trigger based system, script or function call. We denote the condition a *releaser,* because it better suits its role - it releases the rule into execution.

From a formal point of view, the *releaser* is a logical formula.

The action (*exec)* can be viewed as *execution vehicle*, ranging from atomic actions, trough atomic action sequences, to (sub) reactive plans or scripts. The use of either atomic actions or action sequences depends on the AI-engine the agent uses. If the atomic actions are too fine grained, the plans could become too large or hard to design. On the other hand, when too long sequences of actions are used, the agent could be less responsive as a consequence.

In [Picture 4] is a diagram of the SRP model reactive plan.

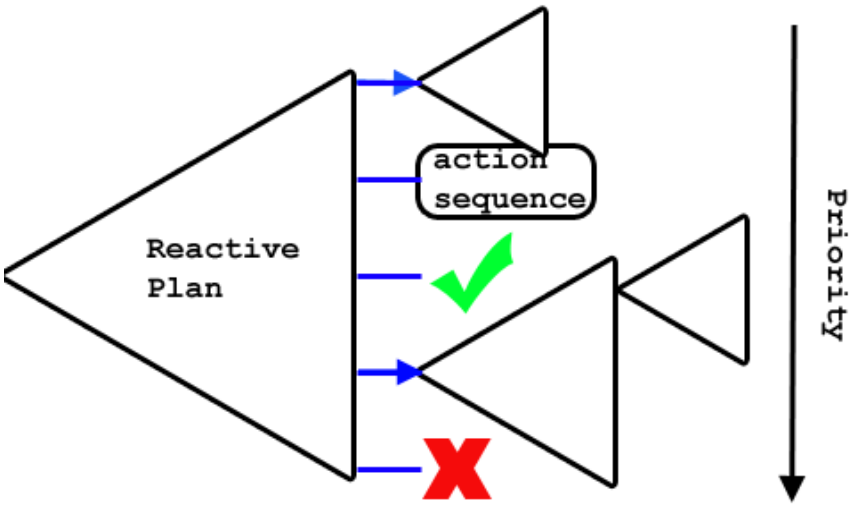


Picture 4: SRP plan diagram

A releaser with an arrowhead on the end, pointing to the *exec* part of the rule indicates, a holding releaser. The rules are ordered by priority, in descending order. In SRP, the *exec* part can be an atomic action or an action sequence.

3.4 Hierarchical Reactive Planning (HRP)

Concept of Hierarchical Reactive Planning (HRP) is an extension to SRP. Actions can be expanded into (sub) *plans* creating a tree-like structure as a result. The bulk of execution is put into leafs of the *be-tree*, which are the atomic actions and actions sequences. The ASM extends into a *recursive algorithm*, expanding the (sub) plans actions on execution and stepping down into them and executing again.

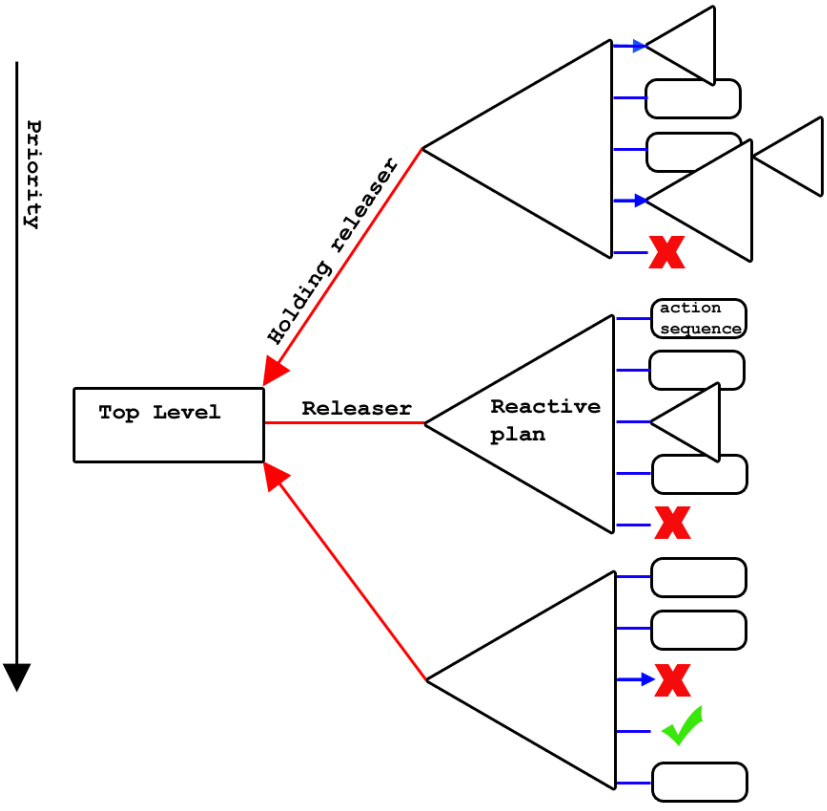


Picture 5: HRP plan diagram

The picture [Picture 5] illustrates the tree-like structure of HRP plans. The triangles represent reactive plans, where action sequences or atomic action are represented by ellipses. The '**X**' sign represents a special action - *"explicit plan failure"* and *check sign* represents a *"explicit success of the plan"* special action. Executing either of these special actions will result in a *fail/success event* stopping the execution of the current level, reporting to the superior level (similar to a "return" in C language).

Holding releasers are lines with an arrowhead.

HRP plans can be used to create *top-level based* behavior [Picture 6], where at the top level is a *top goal* – *"be alive"*. The plans connected via *releasers* represent the *top-level goals* – e.g., "go to toiled", "cook dinner" and "wash clothes".



Picture 6: HRP diagram

The *top-level goals* (and their associated plans) are ordered by priority in the same manor like the rules in SRP. The top-level ASM first checks the releasers for the *top-level* goals, choosing the executed one by highest priority with a holding releaser. Then the HRP version of ASM is invoked.

The structure we used as a template is called a *be-tree* (behavioral tree)[24]. We adopted this term for our structure displayed in [Picture 6].

3.5 Action selection method

In [Algorithm 1] we present a simplified version of the evaluation and action selection algorithm used in *top-level* HRP.

Top-Level:
(1) Take *plan* with holding releaser and highest priority -> *active plan*
(2) If no plan found perform no_op()
(3) If the active plans priority > previously executed plans priority
(3.1) stop previously executed plan
(3.2) AS-*plan* (active plan)
(4) else
(4.1) AS-*plan* (previously executed plan)

AS-plan(plan):
(1) get *rule* with holding releaser and highest priority in plan -> active rule
(2) if no rule found FAIL
(3) if active rules priority > previously executed rules priority
(3.1) stop previous executed rule
(3.2) *execute-action (*active rule)
(4) else
(4.1) *execute-action(* previously executed rule)

***execute-action*(action):**
(1) action is a "plan" -> AS-plan(action)
(2) action is a "atomic action" -> perform(action)
(3) action is a "sequence"
(3.1) get next in sequence (action) -> next action
(3.2) if (next action is after last in sequence)
(3.2.1) perform (first action in sequence)
(3.3) else
(3.3.2) perform (next action)

Algorithm 1: Action selection algorithm

The main idea is to follow the *be-tree* structure where the *releasers* of rules with highest priorities hold. First, a *Top-Level plan* is chosen, based upon holding top-level releaser and maximizing priority. The previously executed plan is halted (external induced *fail).* The search follows the structure, stepping down into the *be-tree.* The algorithm continues recursively until it reaches an atomic action or an action sequence (contains atomic actions). The reached atomic action is executed. If all rules in a plan are inactive, it is considered a fail of the plan, because no action can be executed.

This algorithm ensures that at every evaluation cycle at most one atomic-action (a leaf in the be-tree) is performed. The algorithm performs either an atomic action or steps down in the hierarchy.

The asymptotical complexity of this algorithm is:

O(n∗v∗a)

1. ‘n’ being average amount of releasers in one level of the *be-tree*
2. ‘v’ being the height of the *be-tree*.
3. ‘a’ being is the average time to evaluate a releaser. This can be considered a constant.

3.6 Reactive plan by example

We provide a simple example, which is based upon our scenario in (3.1) – the v-Tom and v-Bob encounter. We provide a reactive plan example for v-Bob in [Table 1].

Priority	Releaser	Action
5	at(shop)	success
4	see(money)	take(money)
3	angry && not(ignoring(v-Tom))	ignore(v-Tom)
2	see(v-Tom) && not(angry) && not(done(chat))	chat(v-Tom)
1	not(dressed) && see(v-Tom)	angry(v-Tom)
0	not(dressed)	goto(shop)
0	dressed	get(clothes)
0	none	Fail

Table 1: v-Bob plan

The plan is pretty simple and does all we need for v-Bob. The logical formulas can be optimized in various ways, considering that rules are checked in order from the highest priority to the lowest and some queries could be answered beforehand.

3	happy	jump
2	not(happy)&& hurt	cry

Table 2: Not optimized rules

3	Happy	jump
2	Hurt	cry

Table 3: Optimized rules

In [Table 3] is an optimized version of the plan in [Table 2]. We can anticipate that the rule considering *happiness* wasn’t satisfied; therefore we don’t need to recheck its condition in the lower priority rule[1].

[1] This kind of optimalization can be done by the releaser engine automatically

The behavior of v-Bob will be pretty straightforward. He gets dressed, if he sees v-Tom before he gets dressed, he will get angry with him and they don’t talk. When dressed, v-Bob goes to the shop, possibly collecting money on the way. When v-Bob sees v-Tom, they talk until one of them ends the chat. After that, v-Bob will continue his walk to the shop collecting money he finds on the way.

It is important to explain the actions *fail* and *success.* A plan can either result in a *success* or *fail*. The *success* has to be explicitly stated, but *fail* can occur either explicitly or implicitly, when no rule is selected as active – no releaser is evaluated *true*.

4 Problems of Action Selection

The approach of SRP and HRP seems pretty straight forward, where choosing the next action boils down to evaluating conditions and walking the *be-tree*. In this chapter, we discuss problems rising from the *action selection method* (ASM) shown in [Algorithm 1].

At every presented problem we provide approaches to compensate for it to gain more believability with as little computational price as possible.

4.1 Interrupting – behavior consistency

More complex behavior may not be possible to boil down to single atomic actions. When introducing more complex action structures, like reactive plans or action sequences, it can be desirable for some of these structures, to be uninterruptible – *interrupt-safe*.

The rule flagged as interrupt-safe cannot be replaced/surpassed by a higher priority rule chosen by the ASM. The interrupt-safe rule stays active, until finished or isn't interrupt-safe anymore. The flag can be understood as an override for releaser checks and action selection, allowing the rule to execute unhindered. The interrupt-flag is therefore a tool to let sequences of multiple actions appear atomic[1].

The example shown in [Table 4] is designed based on behavior of computer game players. It is well established, that running around in the virtual world[2] being low on ammunition in a weapon is risky. Encountering an enemy with almost no ammo, can lead to reloading in mid-battle conditions and that is the worst possible scenario. Reloading regularly, when no enemy is around and when low on ammo, is common, even when potentially dangerous - an enemy could catch upon you while reloading. When the enemy is sighted even while reloading, it is without argument, that first, the reloading has to be finished and then the enemy can be engaged.

Priority	Releaser	Action
4	No ammo	drop(current magazine) take(full magazine) reload
3	see(enemy)	shoot(enemy)
2	Low ammo	store(current magazine) take(full magazine) reload

Table 4: Early reload behavior

The main problem is not to start shooting an empty weapon when an enemy is sighted during *low-on-ammo reloading*. Therefore, the *reload-when-low-on-ammo* action sequence can be flagged interrupt-safe. The agent will finish reloading the weapon and after that, having the enemy in sight, starts shooting.

[1] "Atomic like sequences" are atomic from the execution style point of view, still taking multiple cycles to perform. Atomic actions take only one cycle to perform.
[2] A First Person Shooter (FPS) computer game, like Half-Life2© [10] or Unreal Tournament © [11]

We call this - *behavior consistency* – the agent behaves more consistent in its behavior patterns, not switching wildly between rules.

With responsiveness of the agent in mind, the interrupt-safe action sequences have to be as short as possible, to prevent the agent from being stuck in long *interrupt-safe* sequences, missing out on the world.

When large sequences of atomic actions are used, marking the whole sequence can be considered futile, or undesired. Introducing the interrupt-safe flag not to the actions[1] as a whole, but as a boolean flag for atomic actions (in the sequences) - creating *interrupt-safe zones* [Code 1] within.

```
{[ a1, 0 ],[ a2, 1 ],[ a3, 1],[ a4, 1 ],[ a5, 0 ],[ a6, 0 ]}
```

Code 1: sequence with interrupt safe flag

In [Code 1] is a sequence of six actions {a1, a2, a3, a4, a5, a6} presented. The interrupt-safe zone starts at action ***a2*** and stops at action ***a4*** in [Code 1]. Employing the concept of interrupt-safe zones can render large action sequences more adaptable, allowing a better-structured design of longer sequences.

The interrupt-safe trait propagates upwards trough the entire *be-tree*, meaning when a child node is considered interrupt-safe, its parent is interrupt-safe as a consequence.

It can be argued that the behavior in [Table 4] can be better expressed, introducing more complex releaser configuration not needing the interrupt-safe flag at all. On one hand, it is computationally cheaper to use a boolean flag then a releaser evaluation, and in some cases, the releaser configuration would be too complicated, when multiple interrupt-safe rules are desired. Mimicking interrupt-safe zones behavior employing only releasers can be considered an unmanageable task[2].

4.2 Releaser fault

Lets consider a simple rule [Table 5] in a reactive plan of a v-dog.

1	Next to(meat)	Smile Eat Bark

Table 5: Releaser fault

The problem is not obvious at first sight, but when the v-dog eats the piece of meat it stands next to, the releaser will cease to hold in the next iteration (because the meat is eaten, and the v-dog stands next to nothing) and the last action – bark – is never executed. The rule itself *„breaks“* its own releaser – corrupting its own context. The releaser could fail due to events in the v-world[3], leading the “eat” action eventually to fail. The bottom line is the question: *What will happen with the rule when the releaser is evaluated false in mid-execution of an action sequence, reactive plan or a execution vehicle that takes more then a single atomic action to finish?*

In ABL language [34], two conditions/releasers are used to cope with this problem - *precondition* and *context condition*. The precondition, when satisfied, puts the

[1] The execution part of a reactive rule
[2] The designers point of view taken into account.
[3] Someone eats the piece of meat before the v-dog does

rule into an active state. It is a releaser in a more literal sense. The context condition has to be evaluated true during the execution of the rule's actions. During execution, the precondition releaser is not taken into account. When the context condition fails, the rules execution is stopped – it fails.

We propose a different approach, where a rule can be marked as *releaser-safe*. When marked and its releaser fails, it is ignored by the ASM and considered "true". So even when the releaser is not true, the releaser-safe acts as an override, signaling, the rule's releaser holding. This can be implemented either by a special flag similar to the *interrupt-safe* (4.1) or simply by adding an "OR true" to the releasers logical formula, rendering it always true when evaluated.

It is noteworthy to say, that the releaser-safe trait serves as an override only when the rule is in execution and its releaser fails. The releaser-safe flag doesn't provide an override for a not holding releaser for a not executing rule.

The concept used in ABL language is more powerful then the releaser-safe flag, providing a better but more complex tool. Our approach is less computationally complex, providing a faster option for a speed oriented design.

4.3 Delayed rule activation – sticky rules

Lets consider a rule R1 *t*o be interrupt-safe and executing a action sequence {a1, a2, a3, a4, a5}. A higher priority rule R2 becomes active during the execution of the R1 action sequence. After the action sequence finishes and allows other rules to be become active and be executed, the releaser for R2 doesn't hold anymore. It might be desirable that R2, when its releaser holds, is performed (even when a little bit later).

It could be a response to some high priority event – seeing an *enemy*. We know that the *interrupt-safe* sequence of R1 is short, but the conditions in the v-world that lead to the activation of R2, could change rapidly – the *enemy* hid behind an object.

We need for the R2 rule to be delayed, to stay in the *preactive state* until it can be executed or considered a candidate properly. Therefore we introduce a *sticky-rule flag*, to mark rules which activation has to be delayed if they cannot be executed for some reason (we present other concepts that might delay a sticky rule – e.g., *switching*).

The basic idea is, that the rules when activated, becomes "*sticky"* and even when the releaser doesn't hold anymore, they are considered as a choice for the ASM. Eventually this could lead to a lot of pending low priority *rule*s executed long after their releaser held. Therefore, validity of *sticky-rule flag* should be limited by a *timeout*.

This concept can be viewed as a simple memory, where information about delayed execution during longer *interrupt safe* executions, is stored.

The implementation is pretty straight forward, and the asymptotic complexity for managing the sticky stack is **O(n)** for timeout updates, where **n** is amount of rules present in a plan.

4.4 Fail and success

A *rule* could fail for many reasons – its *releaser* failed and it was not releaser-safe, or the actions taken resulted in a *fail*[1]. A special action is used to explicitly mark *fail* or *success* of a plan – a *fail/success action*.

There are important questions based upon fail and success

- How many times can they occur?

[1] Object was not found, something is blocking the way etc.

- How to propagate them?

4.4.1 How many times to fail?

Lets consider a simple example, where the agent opens a door in front of him. The door is locked, so he tries and fails. The agent's *door-open rule* has high priority and therefore, even when lower priority rules can be active, the agent retries the door-open rule failing multiple times.

To overcome this a *finite amount of fails* should be specified for a rule, and after using them up, the rule will be *disabled* until the plan *resets*. A disabled rule is excluded from the ASM on the current plan. This approach was already presented in the extended POSH (Parallel-rooted, Ordered, Slip-stack Hierarchical) architecture [36].

There is the issue of *resetting the rules/plans* – we need to distinct a *hard reset* and a *soft reset*. The hard reset also resets the fail counters, where the soft reset only resets the rule[1], keeping the counters intact.

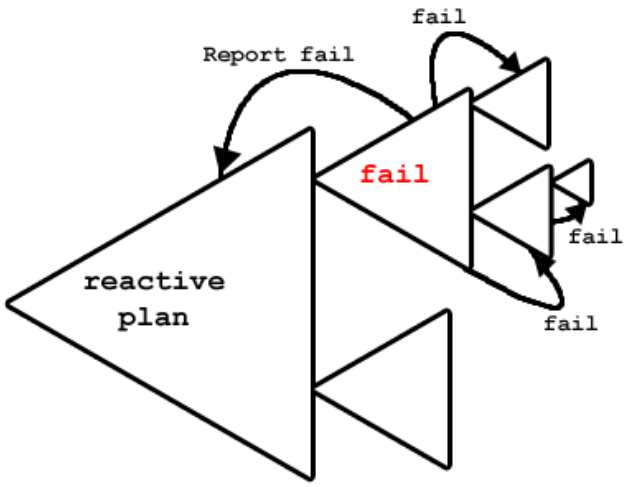
The hard reset is invoked, when the rule is forcefully aborted *"from above"*, where the soft reset is invoked when the rule *fails on its own.* A releaser fault induced forced fail is considered to be "from above".

4.4.2 How many times to succeed?

We propose to apply the same concept as in (4.4.1) to the successful executions, providing more flexibility to the overall concept.

4.4.3 How to propagate failure?

In HRP a *fail* of a plan can happen on any level of the hierarchy, raising the question, "How to deal with *it*?" As seen in [Picture 7], the event has to propagate to the parent of the source, to report what has happened, and to the children plans, to either abort or fail them.



Picture 7: Fail propagation

The issue of proper fail propagation can be seen from the upward and downward point of view.

[1] For instance - resets the positions in the action sequences.

Where the upward direction presents a problem of "how far to go", when sometimes "up to the top" approach may be required[1], when in other cases only a one or two levels up propagation is necessary. For the upward propagation we propose to use a *counter* for the propagating fail, that indicates how far an event has to be reported. An additional "compensation counter" can be introduced into the plan structure, representing how much[2] to subtract from the propagating of an event *counter*.

The downward direction presents a less complex problem, rising only the question "how to deal with the event?" When an event is propagated downward, it can encounter two types of rules that concern it – the *suspended* rules and mostly one *active* rule (it might not be in that current failing sub-tree).

In the case of a *suspended* rule we *abort* the rule – the post-execution responsibilities (like cleaning up objects etc.) are taken over by the *origin* of the forced abort. The suspended rule is *not* given any execution time, it is simply pronounced as finished. The *active* rule is forced to *fail*, but providing execution time to deal with the *fail* (later addressed as Termination in (6.6.5)).

4.4.4 How to propagate success?

To *succeed* is much simpler then to *fail.* A success event can originate only explicitly in a leaf of the *be-tree*, in an action. Therefore it propagates upwards in the *be-tree*, reporting to the containing structure that everything went smooth and it reached the expected result. When a *success* happens on a higher level, the executing lower levels are forced to fail. Suspended rules are aborted.

The concept of "how far go" (4.4.3) can also be employed for success events.

4.5 (Un)biased Random Selection

When multiple rules are of the same priority become candidate to be active, a winner is chosen from their ranks based upon random selection. How unbiased random selection can result in less believable behavior is shown on a simple *v-dog* example in [Table 6].

The dog runs around and occasionally barks – therefore the selection among the rules should tend towards running – it is preferred.

0	True	Bark
0	True	Random run

Table 6: Unbiased random selection

We propose to expand the priority by adding *weight* of rules to the concept.

0 (20)	True	Bark
0 (60)	True	Random run

Table 7: Biased random selection

A rule should be chosen based upon its weight amongst the equal priority candidates. Rules shown in [Table 7] have their weight set. The *"Random run"* rule has a weight of 60

[1] A catastrophic failure on the lower levels of hierarchy could cause this.
[2] Having a default value of 1.

and the "*Bark*" rule has a weight of 20. When both are chosen, the "Bark" rule has a chance of 25% and the "Random run" rule has a chance of 75% to be chosen.

4.6 Rule formalism revised

Based upon the observed problems of ASM, we propose an update on the rule formalism, presented in (3.3).

A reactive plan is a set of rules, where a rule is a octuple

```
{ prio, w, cond, action, flags, sticky timeout, fails, successes }
```

- *prio*, *cond* and *action* semantics don't change, see (3.3).
- parameter *w* denotes the weight of the rule. (Rules with holding releaser are chosen based upon their respective weight to the sum of all the weights of the chosen rules)
- parameter *flags* is composed of flags for *interrupt-safe* (3.6.1), *releaser-safe* (3.6.2) and *sticky rule* (3.6.3) boolean flag. It can be used to store other flags, proposed later in the thesis.
- parameter *sticky timeout* specifies, how long until the sticky-rule expires in the sticky-stack
- parameter *fails/successes* specified how many times a rule can fail/succeed in its execution. When the parameter reaches zero, the rule is disabled from further execution, until the plan is reset

5 Limitations of HRP

This chapter is dedicated to present various limitations discussed in [24] and summarized in [Chapter 2]. We use a scenario to explain the limitations in more detail[1]. There are three main disadvantages of HRP – *lack of intentions, no forward planning and task switching.* The HRP's structures presented in [Chapter 3] are unable to compensate for these disadvantages in a adequate fashion to provide believable behavior as a result.

Some of the presented modifications in this chapter are outlines of the ideas presented in [Chapter 6] in more detail.

5.1 Scenario outline

In this section we will describe the scenario based upon a story of an *r-person (Tom)* who spends his day preparing for winter. This is the story of an ordinary day for *Tom.*

The days are getting shorter and winter is coming. Tom knows, that the winter will be long and hard, so he needs a lot of wood to keep his home properly heated. He gets an axe and cuts down the trees near his house and collects the wood. He knows the axe could get blunt so he takes a sharpening stone along. He also needs something to carry the wood to his house, so he takes a basket. He moves from one tree to another and cuts them down, eventually sharpening the axe, when it gets blunt. After he cuts down the last tree, he takes the basket and collects all the wood, to take it more comfortably to his house. Tom likes to whistle from time to time for joy.

During the day, several events could occur:

- *On the way to the tool shed for his axe, he walks next to his little garden and observes that some beds are dry and need to be watered. After he finishes lumbering, he waters the plans with watering.*
- *If he sees some carrots grown, he collects them when he is done watering, putting them into the basket, he used to carry wood in.*
- *The nature calls, so he goes to the toilet, resuming his work afterwards.*

In this scenario we observe several activities in progress starting with cutting down trees for wood. That is Tom's primary objective for that day, committed to keep his house supplied with wood for the winter. He might add some objectives along the way, depending on the situation.

Tom's activities during the day can be

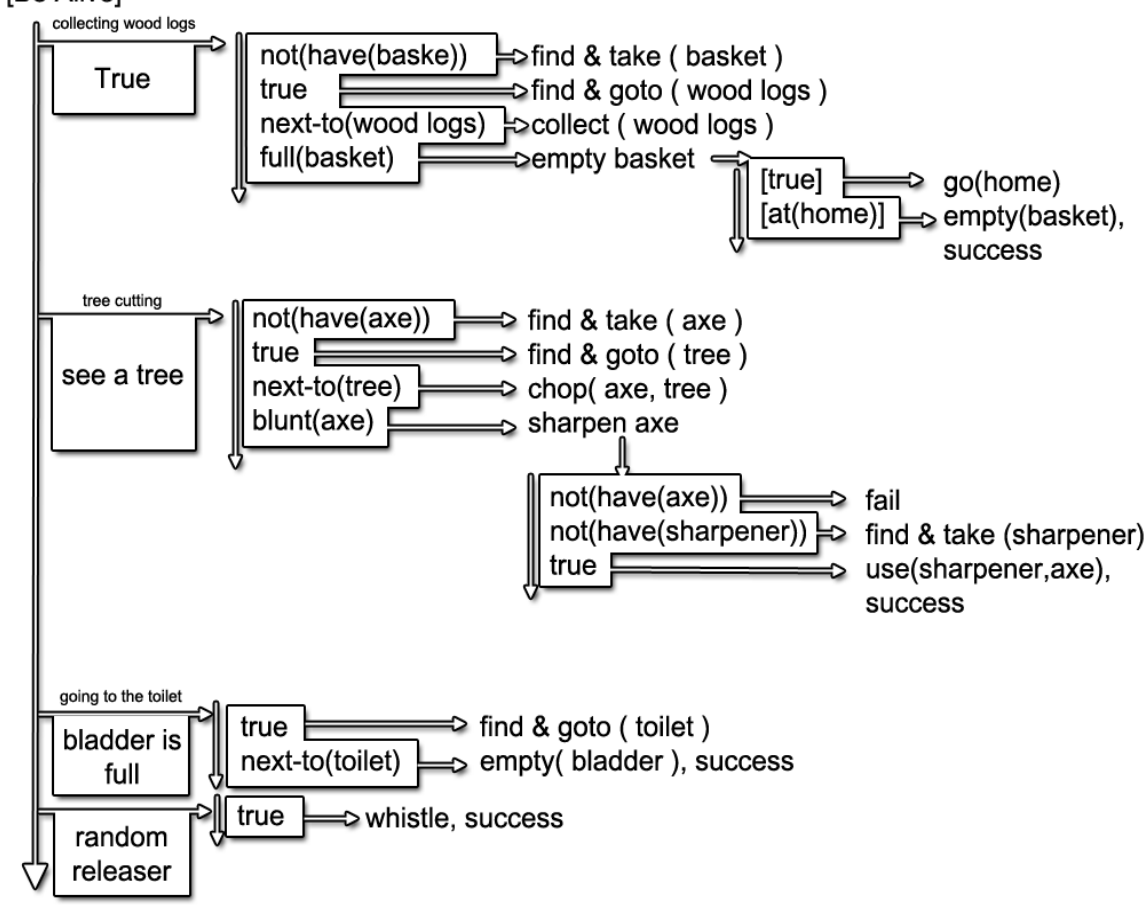
- lumber trees
- collect the wood
- sharpen the blunt axe
- water the garden
- collect carrots
- whistle from time to time
- go to the toilet

When we look upon the mind of v-Tom (r-Toms virtual counterpart), it consists of four top-level goal dedicated to *cut-down-trees, collect-wood, whistle-from-time-to-time* and

[1] The v-world examples present an idea of how agents with HRP model behave, providing a comparable reference.

going-to-the-toilet – providing the setup of the mind to attend to the primary goal of the day. There also can be a top-level goal of *sharpening-a-item*, but it also can be put into the *cut-down-trees*. The outlined *be-tree* [Picture 8] is modeled using HRP.

The outlined scenario is used in this chapter to help explain the limitations of HRP on situations in the scenario. A subset of these situations is adopted in our simulation, providing simple examples, employing the proposed improvements.



Picture 8: Simple reactive plan schema

5.2 Intentions

Intentions can be described based upon the *Belief-Desire-Intention* model (BDI) [12] as a deliberative state of the agent – *desires* he has chosen to commit to, e.g., *"become rich", "have a beautiful garden", "be warm in winter"*.

The HRP model doesn't implicitly include intentions, but as proposed in [24], they can be mapped on the *top-level goals*. With the concept of intentions in reactive planning come several issues that need to be addressed.

5.2.1 Adding new intentions

Adaptation can be considered an attribute of intelligence. A mind, even an artificial, without the ability to modify itself to a certain degree, to adapt to the changes in the environment, fails to produce believable intelligent behavior[1]. We compare real and virtual world example.

R-world: When the r-Tom observes dry beds in the garden, he remembers to attend to them later. When he finishes his current work, and no more important tasks are due to be done, he collects the tools necessary and goes on watering the garden. He might also see some carrots grown, and decide to collect them later. After he finished watering, it might be already too late to collect the carrots, so he decides to do it another time, maybe next day.

***V-world*:** The action selection mechanism of a *v-Tom* lacks the capability of adding, removing or modifying intentions (*water-garden)* to its action selection mechanism. When there is no *collect-carrots* or *water-garden* amongst the *top-level-goals*, they are not considered in the action selection and therefore not executed.

Comment: In the r-world example, r-Tom is committed to two different intentions - garden watering and collecting carrots, where watering precedes collecting and collecting can be abandoned for a specified reason – the day is over.

5.2.2 Choosing alternatives

Choosing alternatives issue can be viewed from two different perspectives – *process based* and *object choice based*. The main difference is that a satisfaction of a *goal* (i.e., w*hat to do?)* can be done by a different approach (i.e., *how to do it?)* or by different means (i.e., *do it by using what?*). A *goal* can be understood as "what has to be achieved" to commit to an *intention.* An intention might require multiple goals to be achieved, some of them optional, some of them mandatory.

The difficulty is that even humans differ in the approach on how they look upon satisfying goals. We tend to look upon things from both perspectives. The difference is in how we view the structure of the process that leads us to achieving our goal.

5.2.2.1 Process based view

A *process-based view* is considered, when different approaches reach the same goal, by a different actions, possibly employing different means (objects).

R-world: The *r-Tom* has to do some watering. He may choose to use the watering can, but he has to walk back and forth to the water tap to fill it. His second option is to use a hose that has to be found first, connected to the water source and then used at the open end. The alternative approaches are different in the way they are executed and differ in the objects they employ.

[1] This depends on the world and the role the agent has to play.

V-world: A *v-Tom* has no choice; he only has a *top-level* goal associated with *watering* where he collects the predefined object and acts based upon the reactive plan associated with the *top-level goal.*

5.2.2.2 Object choice based view

The *object choice based view* makes use of the same set of actions in a (reactive) plan, using different objects.

R-world: When the *r-Tom* decides to cut down trees. He chooses using an axe or a chainsaw, based upon his experience and preferences.

V-world: *V-Tom* has one reactive plan associated with the *top-level goal* of *cutting-down-trees* and he follows the plan, collecting a predefined object and executing the plan steps.

5.2.3 Conclusion

The intention-based issues can be viewed as *modification-based* (adding intentions) and *information-based* (choosing alternative) problems. In [Chapter 7] we propose to

For the modification-based issues, we propose to introduce a specialized type of actions into the model – *modifiers* – action, which can alter the *be-tree* structure. For the modifiers to work, the ASM and the reactive plans have to be able to provide specific functionality for runtime modifications

- adding new rules/plans
- removing plans
- alter plan properties (priority, weight, etc.)

For signaling the request for modification, an adaptation of the mechanism proposed for fail propagation (4.4.3) can be used, where the propagated information contains what modifications should be executed the counter indicating, how far up the *be-tree* this should happen. This also provides that the changes will be contained to one given sub-tree that contains the *modifier* action, not allowing compromising other branches of the *be-tree*.

Exploiting the concept of exceptions[1] should be considered when implementing this concept.

In [Chapter 7], we attend this problem in further detail, introducing the *Intention-add Action (*IaA*)* and *Intention-Goal-Plan map* (IGPmap).

The *information-based* based limitation emerges from the lack of information about the reactive plan. We propose to enrich the reactive plan structure with additional information concerned with the *goal metadata* (6.3) – a list of goals the plan can be used to satisfy. The goal list can be used to do a semantic analysis on which plan to choose (when adding it to the *be-tree* during runtime by modifiers). We also propose to extend the nodes of the *be-tree* so they can accommodate various actions (6.7), possibly a *goal selector* node, which when executed, can choose a plan based the available options for the given goal (possibly fetching them from a central plan repository).

The issue addressed in (5.2.2.2) showed us the need to introduce a list of objects that the plan employs for its execution, providing information for in-plan mechanisms. The *object list* can be considered an explicit enumeration of object requirements by the plan to execute. This topic is described in extensive detail in (6.4)(6.5).

[1] Exceptions in a programming language like C++ or Java.

5.3 Planning

In general, "planning" (in a classical sense) is what *reactive planning* tends to elude, keeping the agent responsive and reactive to its surroundings. However, a degree of planning is necessary to make the behavior of an agent appear more intelligent and believable. Humans expect to see deliberation and planning, to perceive intelligent behavior and see the agent as more self-aware.

The main difference between reactive and classical planning is in the domain they operate on. Where classical planning focuses on single actions, reactive planning settles with entire reactive plans. In classical planning, a result of a single action can be anticipated, knowing its preconditions, effects and workflow, in most cases considering the actions as atomic.

In reactive planning, the *plans* behave in a more "chaotic" way, their course driven by internal and external context. Therefore, it would be not suitable to consider single actions (execution parts of the *if-then* rules) for the *planning*[1].

Following this approach, we can divide the problems for planning into categories

- *pre-execution*
- *post-execution*
- *ordering*

The p*re-execution* phase in a plan is a set of *rules*, which are devoted to create a footing for the other *rules* to perform the "real" *actions* devoted to fulfill the activity (goal) intended by the designer. We call this phase – the *"preparation"*.

It can be viewed in two different setups. *Deep preparation* is localized to the plan hierarchy, related to structures (sub-plans) on the lower levels – a *plan* including *plans.* C*ross plan preparation* is recognized, when the preparation of at least two distinct *plans* overlap.

The *post-execution* issue is tightly related with transitional behavior, explained in (5.4)(cleanup behavior) and [Chapter 6]. *Post-execution* is opposite to *pre-execution,* containing rules, which are devoted to *clean-up* the items and perform actions after the main execution body of a plan has done its job.

Ordering means putting plans in an order, when their successive execution can provide more effective overall behavior, thus making it more believable (*chaining plans*).

5.3.1 Cross plan preparation

Cross plan preparation occurs, when the *preparation* of two or more plans overlaps – their releasers are holding, but at most one of them can be chosen active (and perform its preparation). The others have to wait their turn. Successive execution could lead to ineffective preparation of single plans. Executing in an interleaved fashion, a more effective[2] and believable result can be achieved.

The given example shows, how not considering *cross plan preparation* can lead to a less believable behavior.

R-world: R-Tom goes to the tool shed for his axe. He sees the watering can on the way and remembers he wanted to do some watering later, so he picks the can up.

[1] Planning in reactive planning
[2] In terms of distance traveled, execution time consumed etc.

V-world: V-Tom is executing the *cut-down-trees* plan, first rules dictate to go to the tool shed and collect the axe. He walks by the watering can, ignoring it. When he finished his *cut-down-trees* plan, he starts to search for the watering can.

5.3.1.1 Conclusion

The important question here is, what "preparation" means. Interleaving plain HRP plans execution is not the best way to do it. We can't tell, how the plan will eventually behave when executed, due to the changing context of the environment perceived by the *plan* (by the *releasers* of the *rules*). Our main concern is that the interleaved plan could corrupt the actual active plans execution context, or thanks to events in the environment, need a lot of time to do its work. In conclusion the interleaving plain plans could make the agent appear inconsistent in its execution.

We propose to separate the execution (that might corrupt context, behave unpredictable etc.) from the preparation. The preparation phase being dedicated to collecting objects (preparing in a more literal way), making it easier and more straight forward to negotiate and parameterize the process of *cross plan preparation.* When the preparation phase finishes, the plan enters the execution phase. As a conclusion, the reactive plan can be stripped the rules responsible for collecting the necessary objects. Those objects will be acquired during the preparation phase, or the plan won't execute.

To further fuel the concept, we make use of the already proposed *object lists* (5.2.3) which when introduced into the reactive plan structure, provide the necessary information for the preparation phase.

To compensate for the cross plan preparation constrain, we propose the *cooperative item collection.* When an active plan is in the preparation phase, the ASM might suspend the plan from executing, when a nearby object is needed by suspended/preactive/sticky plan. A suspended plan is "asked" only in the case it was suspended during the preparation phase[1]. This might extensively walk the *be-tree,* where the asked plans, spread the question to the lower level of tie hierarchy, leading to asymptotical complexity of

$$\mathbf{O(n*v)}$$

- 'n' is the amount of plans in the sub-tree
- 'v' the height of the sub-tree

Caching the result of the "need object" query can render the overall execution more effective.

A specific problem arises, when the active plan is not in the preparation phase (being either executed, cleaning up objects etc.) and an object that might be needed by a suspended/preactive/sticky plan is in sight. Due to the unpredictable behavior of the active plan, it might be considered better not to engage the acquisition of the object, it could corrupt its context.

To overcome this, a specific trait can be introduced into the structure of a reactive plan – *focus,* consisting of two values – *actual focus* and *focus limit.* The ASM can query the active plan for its *focus*, resulting in a true/false response – when the plans *actual focus* is above the *focus limit*, resulting in a positive response – the plan is focused and cannot be suspended in favor of cooperative collection of items. When the active plan isn't focused enough, it can be suspended and the needed item acquired. When queried for focus, the active plan can spread the query down the active branch - ask the executing rule. Therefore

[1] A plan in execution phase has all its objects acquired and has no need for further objects

the sub-tree provides the resulting actual focus. When the plan is in another phase, the responsible phase substructure of the plan is queried [Chapter 6].

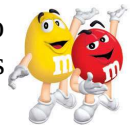
The asymptotical complexity of this approach is

O(v)

- 'v' the height of the sub-tree

Despite all the issues of interleaving and suspending, the most important part is to recognize a needed object. This can be provided by the already proposed object-list (5.2.3). Separating the preparation phase [Chapter 6] from the execution phase can make it easier to combine multiple preparation phases allowing smooth plan execution. Adopting the *focus* feature allows to suspend the active plan in favor of acquiring objects, even when not in the preparation phase.

A drawback of the cooperative item collection approach is that it could lead to *M&M's behavior*[1] – the agent sights an object, walks to collect it, then sights another needed object and so on, following a trail[2].



5.3.2 (Deep) preparation

When we look upon the structure of the HRP, it is obvious that reactive sub-plans, residing deeper in the *be-tree*, can be executed. It is possible that this sub-plan won't be executed at all, but it can be considered wise, to „think in forward[3]".

In plain HRP, it is hard to tell, what object, which sub-plan, needs.

R-world: R-Tom goes to cut down trees and he knows that the axe he uses could get dull during the day. So he takes a sharpening stone just in case he needs it.

V-world: V-Tom executes the *cut-down-trees* plan. Collects the axe and goes to the forest. When the axe becomes dull, his sub-plan *sharpen-the-axe* becomes active. He goes back to the house and takes the sharpening stone, comes back to the woods, sharpens the axe and continues falling trees.

5.3.2.1 Conclusion

Humans perceive this kind of behavior as a mark of experience with the given activity - thinking in forward. When doing something for the first time, we don't consider all the possibilities that could happen and omit something we might need. When this situation occurs, we tend to learn from our "mistake", anticipating the need for additional "in case it might happen" items.

The already proposed concept of *object lists* (5.2.3) can be used here with great success. A designer could put „experience" into the object list, not only including the object necessary for the current execution level of the *reactive plan*, but also some of its *sub-plans*, preparing for "what might happen".

A different approach might be employed to simulate biologically plausible behavior, where every plan at every level of the hierarchy, contains only the minimum necessary object list for its execution and when a plan on a lower level succeeds, the "experience" is propagated upwards in the hierarchy, uniting the object lists from the successfully executed lower level with the list in the higher level. Thus adapting the behavior to the conditions in the *v-world* providing a simple but recognizable learning mechanism.

---

[1] Based on the M&M's candy.
[2] This can be exploited to lure an agent into a trap at the end of the trail.
[3] This can be rephrased as „Think in deep" to better describe the process.

The drawback of this "simple learning" is that it provides a "unforgettable lesson" - there is no mechanism, how to "erase" the specific objects from the object list, rising the questions about how to design the mechanism of "forgetting". The approach, one hand providing the lower level plan with the items needed, on the other hand, taking the possibility to choose alternatives (5.2.2) is beyond the scope of this thesis.

It is also possible, that the plan could perform an analysis before executing his preparation phase. The object lists of lower level plans acquired and put into the object list and including them into its own preparation. The objects can be marked as optional, not forcing the plan to fail, when not found.

This approach is rather slow, because the whole sub-tree of a plan needs to be traversed, providing a complexity of

$$O(n*v)$$

- 'n' is the average amount of plans in one level of the sub-tree
- 'v' the height of the sub-tree

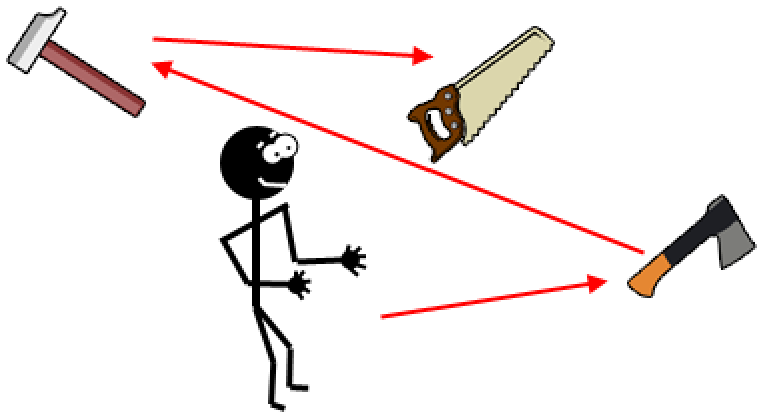
5.3.3 Collecting objects (effectively)

The process of acquiring key objects for plans is an important part of the execution. Most of the plans need objects to successfully influence the *world*. Without the proper means (objects) there is no way to fulfill the plans goal – a *cut-the-trees* plan can't succeed without an axe or a similar item.

R-world: When the *r-Tom* is going to cut trees, he knows he needs an axe, a sharpening stone (the axe may get blunt during the day) and a basket to carry the wood. He looks around, choosing a possibly optimal way to acquire the objects. If the axe is missing, he goes to the tool shed and picks it up there. When equipped, he goes to the forest to cut down some trees.

V-world: *V-Tom* executes *plan rules*, dictated by their priorities. Collects the *v-objects* following a given order of rules dictated by their priorities. He always follows the same order, even when it is the least effective way to collect the items [Picture 9].

Comment: This can be viewed as appetitive behavior, where the agent is put into a situation where his actual goal dictates a needed set of objects the agent needs to collect to be able to continue.



Picture 9: Item collecting

5.3.3.1 Conclusion

When we think of human like behavior, we could see this problem in two perspectives - when all of the needed objects are in sight and there is a reasonable amount

of them to perceive and be able to reason swiftly about the order in which to pick them up. The other perspective is, where multiple possibly hidden objects are involved, creating a more complicated situation, forcing the mind to do a complex reasoning and search. The topic of affordance and direct perceiving is discussed by James J. Gibson in [43].

For the v-human it would be more effective to enrich the reactive plan structure with information about the means, which are needed to reach the desired goal[1]. A set of objects (5.2.3) associated with the goal (reactive plan) could be considered even biologically plausible. When a human intents to follow a *goal*, he prepares the necessary tools, making an internal checklist of objects - "making a cupboard" → box of nails, a hammer, 10 wooden planks.

We discuss the *object lists* in more detail in [Chapter 6].

The reasoning about the perceived object in sight can be performed in the preparation phase, providing an optimal way to collect them.

5.3.4 Search for objects with (sharp) timeouts

We all search for things from time to time. Lost keys, missing glasses or misplaced tools. In most cases we put a reasonable timeout to the search – 10 minutes, 20 minutes and then we loose all patience and our attempt fails[2].

The *v-agents* can't spend all their time searching for an object, so we also put a timeout on v-agents searches. The timeout is a sharp exactly defined amount of time, the agent can use for the search of an object.

R-world: *R-Tom* remembers that he left his axe in the tool shed, but he can't find it there. He starts to search the house, when his patience (internal timeout) runs out, he considers his attempt to find the axe as failed, but on the last moment, he spots the axe in the living room. He takes it and continues on with the chosen activity.

V-world: *V-Tom* can't see the *v-axe* and starts the search. His timeout is set to 5 minutes. When the timeout expires, he stops the search, even when the object is in sight and only few steps away, considering the timeout expired and failing the search.

5.3.4.1 Conclusion

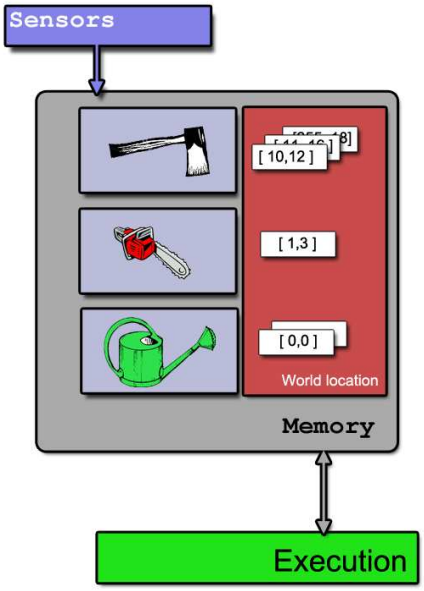
A simple build-in condition when searching can be added to always soften the timeout - `"see(object)"`, meaning, when the object is seen, it will be collected regardless of the timeout. This condition can be hidden in the search and acquire algorithm employed in the preparation phase. The softening of conditions was proposed in [24]. We propose to include this mechanism implicitly during the search performed on an automated basis during the preparation phase.

To search for an object can be a tricky task. Humans seek first at places they think the object might be at – trying to recall *"where have I seen it for the last time"*. After using up all the options, we start to run out of patience and perform random searches.

It would be wise to exploit this "simple" mechanism, implanting a *place-object memory* into the actor's brain [Picture 10], providing the search with some data where to start looking. The agent also should collect relevant data (location) during regular sweeps by (visual) preceptors to keep the memory up-to-date. At this point an animat's brain can be overloaded when put into an environment containing massive amounts of objects.

[1] Goals can be reached by executing a plan.
[2] And when all hope is lost, the object "magically" appears somewhere we are certain we looked for it.

Therefore a filter is needed, where the list of objects can be used as a filter. The filter would be a unification of all top-level object lists of "interesting objects". Even humans don't remember all what they see, they focus and filter things of importance to them and their goals.



Picture 10: Simple Object-Place memory

5.3.5 Plan chaining

Humans tend to act continuously, preferring tasks with similarities, like common tools or common places, making the overall behavior more effective. When executing various behaviors that share common places and objects, ordering them to heed these resemblances can make the overall execution appear more smooth and effective.

Remark: We extend our scenario by the goal of going to the shop that is situated in a village 10km away[1] from Tom's house.

R-world: V-Tom finishes watering his garden and considers a choice to go to the shop to buy some food or collect carrots in his garden. On one hand, the carrots can wait, but the shop is a 10km walk from his house but will be closed after 18:00 (6PM). Depending on the emptiness of his fridge, he chooses his next activity, probably first plucking some carrots and then going to the shop.

V-world: When the *v-Tom* finishes watering the dry beds in the garden, he chooses the next plan based upon priority, going to the shop[2], then coming back and collecting carrots.

5.3.5.1 Conclusion

The accounting factors that "link" plans are the *location* and the *employed* (required) *objects* – more factors in common and shorter distances make the link stronger. Therefore, chaining of plans is considered when a plan is either finished or suspended and others from preactive plans compete for the "open position" for execution.

[1] If a 20km walk isn't discouraging enough, double or triple it.
[2] The visit to the shop has a higher priority then collecting carrots.

We can specify the object-overlapping factor by the already proposed object list (6.4), which provides the chaining mechanism with information about common objects. A larger common intersection of objects strengthens the link.

Knowing “where“ (location of execution) by a plan can be used to chain plans even more effectively. Object requirements might not be sufficient to create plausible chaining. Two different plans could share objects without any other association, performing the execution on the opposite sides of the v-world. Therefore growing distance is a weakening factor for the link.

A location of plan execution could be easily perceived as a rectangle enveloping all the “places” where the agent performed rules contained in the plan. To provide a more customizable design, rules could carry a *track flag*. When a rule is flagged with the track flag, its successful execution would be considered a point of interest for the *location of execution* for the plan. The *track* flag can be incorporated in the *flag parameter* of the revised rule formalism (4.6).

There is a variety of approaches how to employ this concept

- considering only equal priority rules, providing a bonus to their weight
- applying the chaining mechanism as a bonus for priorities and weights for all preactive rules, allowing rules to be selected, thanks to the affiliation with the currently finished or suspending rule.

The *bonus function* should be designed with respect to the *v-world* it is employed in. To provide even more flexibility, the bonus function could be introduced into the reactive plan structure, providing a *per-plan* mechanism – the suspending or finishing plan could influence the decision process regarding his successor.

An explicit list of “plan links” with their modifiers (to priority and weight) can be specified per-plan or in a global table, providing an alternative to a bonus function.

The concept can be applied to any level of the *be-tree* hierarchy, where rules are competing to be selected.

5.4 Transitional behavior

In a dynamic environment, a situation can occur, where a *reactive plan,* could be interrupted in mid-execution by another *plan* or *action* with higher priority. In some cases, the observer awaits a smooth transition from one to the other behavior. This topic was also extensively addressed in [15]. In plain HRP expressing transitional behavior can be almost impossible to achieve, because the plans don't have any knowledge of other plans that could be succeed or surpass them.

R-world: The nature calls while *r-Tom* is watering plants. He puts the watering can down and goes to the toilet. When done, he returns back, picks the can up and continues watering.

V-world: The need to go to the toilet is an internal drive that manifests itself by a holding *top-level releaser.* The associated *plan* executes and *v-Tom* goes to the toilet holding the watering can in his hand. He puts it down next to the *v-toilet*. After he answered the call of nature, he resumes his *watering plan.*

Comment: In [24] this kind of human behavior is described as *cleaning behavior*. There are two kinds of that behavior – a *pure* form and a *transitional* form.

A *pure* form means, the item/items used during an activity are *cleaned up* – returned to their original places, or simply stored somewhere. This topic is addressed separately in (5.5).

The transitional form is used when one activity has to be suspended by another activity, with a higher priority at the moment. The activity is discontinued in a consistent way, so the suspending activity can be engaged without obstructions and the suspended activity can be later resumed.

The biological plausibility can be easily observed, we (humans) tend to do the *transitional behaviors* when we have the time for it. We tend to leave our suspended activities in a consistent state, to be able to resume them easily.

Lets consider simple real life scenario, where a person is watching a movie. Suddenly the phone rings. She/He pauses the movie and picks up the phone. This can be understood as a transitional form. But when a fire starts, or a plane hits the building, he/she doesn't think about pausing the movie, she/he just flees.

So we can assume, there is a factor of *urgency* that has to be taken into account.

5.4.1 Conclusion

We use the term *switch behavior* for the transitional form of transitional behavior.

A solution to the *transitional* form has been proposed in [15], based upon a square matrix (N x N), where the pairs of behaviors are specified, applied when one behavior switches to another. In general, this idea provides a suitable solution to the problem at hand.

The main problem, however is in the unpredictable fashion the reactive plan execute. The transitional behavior can be invoked at any time during plan execution, making it difficult to expect a specific state of the plan (e.g., right hand holding the sword, left hand the shield) – it could be in the middle of collecting objects needed for its execution or during cleanup, when a portion of the expected objects is already returned to their places. This can provide faulty transitions rendering the approach problematic or not functioning at all.

We propose using a similar approach, where every plan is equipped with a *default transitional behavior* (DTB) and a *related transitional behavior* (RTB) – a database of

transitional behaviors between RTB owner and other plans. The DTB is used when no pair in RTB is found.

There is also the issue of *urgency* for the *switching* described earlier. The *urgency* can be viewed as a floating-point number from <0,1>. Extending the RTB by ranges of *urgency*, where a *switching behavior* is assigned to every given range, can lead to more versatile and also more believable behavior – *v-agent* could store the items, he collected, in his backpack when switching to another plan, when no imminent danger is around or throw the items in the direction of an incoming *v-enemy*, who is posing a great deal of danger.

We propose to employ a separated phase for entering and leaving the suspended state. The separated phases can employ consistency checks, and switching of behaviors can be manipulated during these phases. As a conclusion, a specific phase should be dedicated to the suspended state.

Later, in [Chapter 6] we address this issue in more detail, explaining the problems that should be taken into account when employing this concept.

5.5 Cleanup behavior

For humans, it is common to perform a *cleanup* after they finish an activity. By this, we tend to keep our view of the world in a more consistent shape, not needing to update the "picture of the world" every time we finish an activity. Possibly having a biological reason – we clean up items to the same place over and over and our memory about it reinforces, making it easier to recall the place where to start a search for that object. Shortening the search time makes further activities more effective, not doing unnecessary searches for misplaced items.

The cleanup behavior raises two important issues that when unattended can render the behavior of an agent less believable.

1. Cleanup of objects after a plan has finished
2. Cleaning up objects that are needed by other plans

In the next example, we use a different scenario, then outlined in (5.1).

R-World: Tom finishes repairing his bicycle, using a screwdriver and a hammer. He goes to the tool shed and puts the screwdriver back, but keeps the screwdriver, because he knows he will need it to repair other things in his house.

V-World: V-Tom finishes repairing his v-bicycle, goes to the tool shed, puts the v-screwdriver and v-hammer back to their place. In the next instant (the plan associated with hanging repairing items in his house becomes active) he picks the screwdriver up and continues.

Comment: A worse thing could happen to v-Tom – the rule acquiring the screwdriver will be executed later, so he walks away, collects other objects needed for the repairs and then comes back to get the screwdriver.

In HRP, the issue (1) can be modeled by adding rules to the end of the plan. This has two major drawbacks

- The objects will be cleaned in a predefined, possibly ineffective order
- When the plan fails, before the objects are returned, they might be never returned - no plan will need them and therefore none will return them

5.5.1 Conclusion

We propose to distinguish the *cleanup phase* from the execution phase making reactive plans more modular and adaptable. Separation allows parameterize or partially/completely omit this phase. As a result a more smooth behavior can be observed. Common objects for various plans can be excluded from cleanups, providing a safe[1] and believable behavior. Parameters for this phase can vary, from a timeout, which limits the cleanup execution, to providing a drop point for certain types of objects[2].

The issue of excluded objects is straightforward – a finishing plan shouldn't disrupt the execution context of a suspended rule. It can be considered prudent to keep objects that might come in handy for preactive or sticky rules, to spare the search and acquisition of that object.

Suspending the *cleanup-phase* can also be used by higher-level *reactive plans* to collect items used by the *plans* on lower level of the hierarchy, performing an "all used items" cleanup at the end. This could prevent reacquiring of items, when a *plan* on a lower level of hierarchy is executed repeatedly.

[1] Plans with common objects don't disturb each others context by dropping them somewhere.
[2] Similar types of objects can be stored in one location, although found dropped somewhere.

In HRP, a failing plan might not take care of the acquired objects it collected, when the execution didn't reach the rules responsible for those actions. A cleanup action could fail, due to unreachable destination or any other problem. The question arises, who takes care of the *"forgotten objects"*. The parent node (in the *be*-tree) should take care of objects collected and forgotten by its children nodes. As a conclusion, the agent should take care of cleaning up objects forgotten by its top-level plans.

Introducing a *specialized* type of plan can compensate the *"top-level forgotten items"* issue. Having a priority and a specialized releaser, which holds, when the agent is near a location of a forgotten object, performing the cleanup behavior. The list of forgotten objects has to be regularly updated, to avoid cleaning needed objects. We address these type of node the *garbage nodes*.

Another important issue is to update the cleanup list of the actually cleaning plan to take new preactive plans into account. It can be done either adaptively, only adding the new preactive plans to the exclude list, or create the list anew. These updates have to be done also to the resumed plans, to ensure consistency.

Creating the list anew can lead to behavior artifacts, where the object is cleaned up, and the instant, the agent reaches the destination to drop the object off. The plan that needed that object becomes active, rendering the object needed and excluded from the cleanup. When the agent leaves the area, the plan needing it becomes inactive rendering the object available for cleanup.

5.6 Behavior aftermath

Human behavior can end having various results – *fail, success, partial success, Pyrrhic victory* [38] etc. – affecting the internal states of the mind – feeling anger, frustration, joy or motivation.

The HRP model can only handle *fail* or *success*, lacking the ability to express any *consequences* based upon the *plans execution result*.

R-world: *R-Tom* waters his garden using a *watering can.* The can is pretty old and its handle breaks of, rendering the can useless. His attempt to water his garden *fails* so he gets angry, kicks the can and curses for some time.

V-world: For v-*Tom*, the *v-watering can* object is rendered useless during *watering* and the action employing it *fails* and the entire plan fails (there is nothing to water with). The ASM notices the *fail*, possibly disabling the plan from further selection and continues on with another plan. There is no mechanism to process *fails* or *successes.*

Comment: A behavior aftermath is important in human perception of intelligence, implying the awareness of the executed plan and its *results.* The aftermath can also be used when the designer wants to modify the internal clockwork of the agents mind – emotions, moods etc.

5.6.1 Conclusion

We propose distinguish a specific "aftermath" phase, to provide the necessary expansion to deal with the *result* of a given *plan.* The *aftermath state* would provide a more adaptable and modular design and make it available for a *result* to be processed.

To simplify things, we consider only *fail* and *success*, but a floating-point value could be easily used to specify the amount of *fail/success* in the aftermath state.

The result of a plan could be modified from a true(success)/false(fail) logic into a floating-point logic, where the result is a floating-point number from the range <0,1>, fail represented by 0 and success by 1. The resulting value[1] would indicate, how much of a fail/success the plan was[2].

In the aftermath phase, the result could be evaluated, choosing a behavior based upon the mapping to a range [Table 8].

< 0, 0.2 >	Get angry
(0.2, 0.8 >	Normal behavior
(0.8, 1 >	Feel joy

Table 8: Floating point mapped success/fail behavior

[1] Ranging from "miserable failure" to "gargantuan success".

[2] When shooting at a v-enemy, the amount of accurate shots can be reported as a measure of success by the rule that explicitly invokes the end (fail/success) of the behavior.

6 Improving Reactive Plans

In this chapter, we present and discuss our structural improvements to reactive plans and their internal mechanics to better cope with problems and limits presented in [Chapter 4] and [Chapter 5]. We try to look upon the reactive plans from a more object oriented perspective, on one hand putting a lot of additional information into the plan structure, to aid the parts of the *action selection*, on other hand changing the way the reactive plans internally behave by introducing a *phase-like* architecture to them, but keeping the overall concept intact[1].

Our main approach is concerned with transforming reactive plans from simple *if-then rules containers* into more capable subjects with internal logic, exploiting techniques from object oriented programming languages (C++/Java), giving the concept the advantage of a modular design.

When we look upon the reactive plans from the SRP's point of view, the plan itself is an only by priority *ordered set* of *releaser-execution* pairs. The HRP introduces hierarchy to the concept, but the basic idea stays unchanged. When a v-agent's virtual brain is equipped with this ASM based upon pure HRP and put into a dynamic and unpredictable environment, it gets difficult to perform any build-in or external (automated) analysis on the current state of the reactive plan[2].

The limits and problems presented in previous chapters are a manifestation of lacking overall (external and internal) control over execution and virtually no auditing possible from ASM's side. Absence of information flow from the plans to the ASM can lead to difficulties in synchronization of plan execution to maintain either continuance or consistency.

We have to keep in mind, that the timely fashion of execution should be comparable to HRP. The other important issue is to maintain a layout that can be easily understood and exploited by AI-designers.

6.1 A different perspective

Lets take a more detailed look upon a reactive plan from a different point of view. The basic idea of a plan is to satisfy a *goal* of the v-agent in the v-world. The goal can range from simple things like *weeding a garden* to more complex goals like *capturing a building,* consisting of sub-goals - *clear room*, *cover staircase*, *rescue hostage*, (associated with plans on lower levels of hierarchy).

For every goal, there are *means* – objects – that are necessary to attain it. Most of the plans cannot be successfully performed without a set of objects – e.g., *v-hammer* for *hammering nails*, *v-saw* or *v-axe* for *tree felling* etc. An *object-based perspective* can be considered biologically plausible – humans tend to associate certain objects or types of objects with activities, having difficulties using different sets of tools for the same tasks.[3].

In third perspective is *execution-based*. Considering a simple human activity - "*nailing a nail into the wall*" - perceived as an *intention*, with an associated goal of "*hammer-nailing*". The earlier mentioned *object-based perspective* dictates, that we need a hammer (or similarly brutal tool) to hammer the nail into the wall. Without the hammer, the given goal cannot be accomplished. Lets assume we found the hammer and got it in our hand. Now we have satisfied the *initialization preconditions* of our goal and

[1] **If-then** rules are still the core of execution.
[2] A HRP plan can be considered a blackbox with unpredictable internal behavior.
[3] On the other hand, humans are good at improvising – e.g., hammering a nail with a shoe

can proceed to the actual *execution*. Finding a nail can be either considered a part of the initialization or actual execution. Considering it a part of the execution, where first finding the right nail, then going to the wall and using the hammer on the nail driving it into the wall is the actual execution. If the nail bends, we get another one, until it is properly anchored in the wall. As a conclusion, we have reached our goal, and nailed it. Now we need to put the hammer back, leaving it in next to the wall seems inadequate. We are finished and ready to do something else.

Lets explore this scenario a little further. Two things could happen - a fire could start, or the phone could ring. If the fire starts, we really don't care about anything else and simply run. But when the phone rings, we stop hammering, put the hammer next to the wall, or into our pocket and go answer the phone. It won't ring forever.

As seen in this simple scenario, the activity committed to the intention of *"nailing a nail"* and goal of *"hammer-nailing"* went trough several phases - *initialization*, *execution*, possible *transition* to another activity, *transitioning back* and finally *cleanup*. It may be hard or impossible to simulate this scenario using SRP/HRP with adequate results.

We adopt the idea of an activity going trough separated *phases,* and introduce *statefullness (phasefullness)* to the reactive plans. We addressed the need for distinguished phases earlier, in [Chapter 5].

6.2 Blueprints and instances

In later discussion, we distinguish two types of reactive plans – *blueprints* and *instances.*

A *reactive plan blueprint* (RPB) is a reactive plan that is not executed by any agent. It is a template for a plan that can be stored in a *central repository*.

Moving a RPB from the central repository of plans into an agents *mind*, making it a part of its *execution process* or ASM is called *instancing* – resulting in creating a *reactive plan instance* (RPI).

The instancing process is essential, making plans available for different agents, or usable on different levels and sub-levels in the *be-tree*.

Internal configuration of a plan corresponds to the overall *state* of the plans execution - positions of atomic actions in action sequences, sub-structure states, disabled rules etc.

The issue on how to instantiate large RPB structures arises, presenting two possible approaches. The RPB structure could be instantiated completely at introducing it into an agents mind, or it could be done "on demand".

6.3 Intention and goals metadata

Based upon the *belief-desire-intentio*ns (BDI) model [12], we understand the *intentions* as a *selector* for certain goals to be followed. In HRP architecture, the *goals* were associated with *top-level behaviors*. We propose to expand the reactive plan structure, adding a meta information of goals to a plan.

The meta information added to a plan being a *list of goals* the plan can be used to accomplish when successfully finished. When committed to an intention a set of plans will be chosen based upon the demand for given goals, allowing to search for alternatives.

```
plan("gardening") -> goals {"water", "weed"}
```

Code 1: goal list of a plan

We can view the commitment to intentions as satisfying a set of goals associated with it in a given order [Code 2].

```
Intention (“have a beautiful garden”) -> goals {“water”, “weed”}
```

Code 2: intentions to goals

6.3.1 Conclusion

Specifying a *set of goals* for every plan can be used to overcome the limit of *choosing a process based alternative* (5.2.2.1) when committing to an *intention*. The list of goals can be used to choose plans from the plan library or a central plan repository, adding plans based on a goal or intention specified request. This mechanism presents a simple but effective solution to the *adding intentions* constrain (5.2.1) when used in conjunction with later proposed extended action (6.7), which invoke the specific requests.

6.4 Object set

The idea behind the *object set* is simple – a list of explicitly specified objects, given by their unique identifier. To provide more power to the concept, we view the *object set* as a disjunctive normal form in boolean logic [Code 3]. The *object list* is a conjunction of objects and the *object set* is the disjunction of object lists.

```
(Obj1 & Obj2 & Obj3) || (Obj1 & Obj4) || (Obj6)
```

Code 3: object set formalism

The object lists in the object set are ordered by their *importance*. The requirement is to satisfy the object list of higher importance as soon as possible, where the lower importance lists are possible options to be satisfied. During object search, time factor is of essence, therefore satisfying a lower importance object list is considered a success and the search can end. Therefore, every list has to be composed in such way, to satisfy the requirements for objects of the plan on its own.

To express the need for timeouts during searches, every object should have a timeout specified. An expired timeout for at least one object renders the list “failed”. The formalism update is specified in [Code 4], where empty brackets represent an “infinite timeout[1]”.

```
(Obj1[0:30] & Obj2[0:45] & Obj3[1:00]) || (Obj1[] & Obj4[10:00])
                                || (Obj6[0:10])
```

Code 4: object set formalism with timeout

Also, specific flags can be introduced into the formalism allowing the designer to better specify, how the search algorithm should treat certain object lists – preferred, optional or last resort[2] list etc.

We provide a simple version of an update algorithm in [Algorithm 6] to illustrate how the object sets should be used in conjunction with a search method.

[1] Using an infinite timeout should be omitted.
[2] When all the other lists have failed.

1.1 Update status of object from already collected items (in the inventory)
1.2 Check for a satisfied *list* with the highest priority that is not marked *failed* and return it to the search method
1.3 Check for expired timeouts
1.4 Mark every list with at least one object with a expired timeout as *failed*
1.5 Check objects in nearby to the actor
1.6 Choose the object with the shortest route (in respect to path finding) that is in a *list* that has not been marked *failed*
1.7 return the *target* to the search method

Algorithm 6: Object set update

Adding partial ordering and prioritizing objects in a single list can further improve the object set semantics.

To provide more flexibility to the concept, a primary/secondary flag can be added per object, along with the already present timeout. A *primary* object is mandatory for the given *object list* to be satisfied, where the *secondary* object is optional.

6.4.1 Conclusion

Enriching the structure of reactive plans with the *object set* can provide more information about the plan – an explicit listing of object requirements for proper execution. This information can be later processed by various mechanisms aiding searches and identifying plans requirements.

6.5 Object classes

We (humans) can look upon objects in different ways, asking us various questions about

- its **traits** – *is it sharp?*
- its **purpose** - *can it be used for gardening?*
- how to **use** it – *can it cut things?*

This simplified view of human object perception and semantics is in our opinion considered sufficient for a v-agent. To introduce the semantic information about an object, we introduce *object classes.* This concept was inspired by the academically unpublished game mechanics and structure for objects used in the game Sims© [1]. The so-called *smart objects* [36] publish information to their surroundings - which *needs* of the in-game agents they satisfy [37] - providing an *advertising concept,* allowing the agents to search for objects that suit them and their *needs.*

The *object class* can be viewed as a *description tag,* answering different questions about the object – its properties, its function or how to put it to use. When we look upon a v-axe – there can be many v-axes in the v-world, each of them identified by a unique identifier, but they share the common trait of being an *axe.* Therefore, the common object class is “axe“. A v-knife, beside of having the object class of “knife“, could have object class of “has-blade“. This trait is also shared with the v-axe.

The „axe“ and “knife” classes define the *object type,* where the *”has-blade“* provides *property information* about the given object. We also could think of other classes, specifying other information like *”cut“* and *”chop“* denoting available *interfaces*.

The object classes can be used primarily to distinguish objects with desired specifications, filtering the objects we desire – those we want to use for *cutting* things and *having a blade.*

We propose a simple list with a globally specified *timeout*, formalized in [Code 5].

```
(ClassA & ClassB & ClassC)[10:00]
```
Code 5: object class list formalism

The list only specifies, which classes have to be satisfied, not concerned about specific combinations of classes per object. A more complex formalism could specify objects with specific class sets, similar to (6.4).

```
(A & B & C)[10:00] + [ M,B,C ], [ Q,D,A ],...
```
Code 6: object class list formalism

In [Code 6] a *global set* with a timeout is specified on the left side of the plus sign. This list has to be satisfied until the timer "[10:00]" expires. The sets of classes specified on the right side of the plus sign are to be satisfied per object. For every set, at least one object has to be present (in the inventory), having all the listed object classes. The whole *extended class list* is satisfied, when all the *global classes* and *per object sets* are satisfied.

The *primary*/*secondary* specifier proposed in the previous section can be employed, to further specify requirements on *object classes* – distinguishing mandatory and optional class specifications.

6.5.1 Conclusion

Enriching the structure of reactive plans with the *object class set* can provide more explicit information about the plan – what object classes might be needed for execution. Putting more semantic information into the plan's structure provides a more general approach for specifying needed objects[1]. This concept also allows employing more general searches, where the agent is concerned with a property and not the actual object it uses.

Adding this general information to the plan's structure can provide the needed semantic information about the plans requirements.

6.6 State full Hierarchical Reactive Plans

A reactive plan in HRP or SRP can be viewed as a *stateless* form of a *plan*. We propose adding explicit *states* - *phases* to the structure of reactive plans, creating a *State Full HRP* (SF-HRP), rendering the plan's execution structured and therefore putting it under better control.

The idea originated from observing the behavior of average plans and identifying specific stages the plan goes trough. We also used our understanding on how humans approach the creation of reactive plans. The designer of the animats minds is an important factor to consider - the human understanding on "how to do" the modeled activities. Therefore, the plans structure should mimic the decomposition of a problem by a human mind – putting it into distinct phases. The SF-HRP model might be considered more biologically accurate, in respect to the human approach. We conducted few simple

[1] When an agent is attacked, a self-defense plan can require an object having a blade, not caring about the actual specific item.

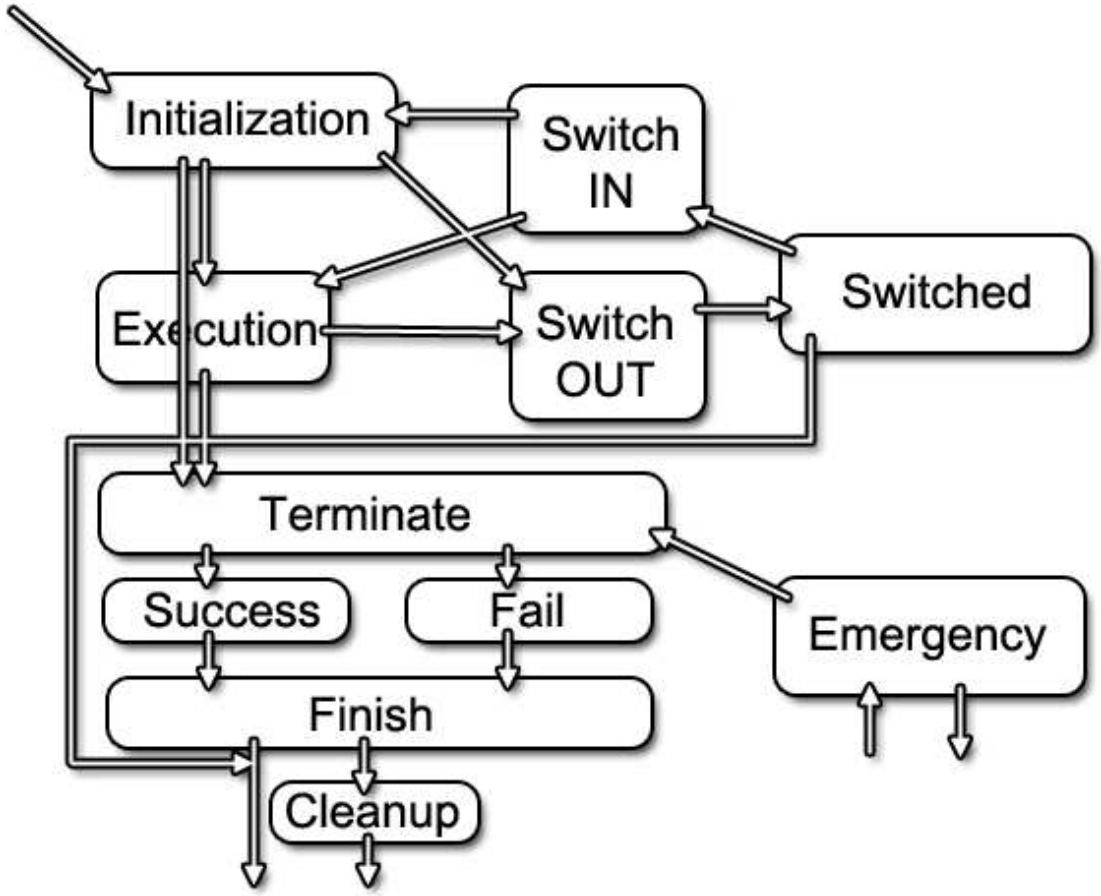
experiments, asking various people to describe activities in detail and observed the common patterns in decomposing the activity at hand, identifying phases an activity may be thought to go trough.

In the SF-HRP concept, the HRP model is situated in the *execution phase*. The other later discussed phases are:

initialization, termination, finish, cleanup, switch in, switch out, switched, emergency

Each of those phases is dedicated to fulfill a specific role in the overall execution. The phases allow the control mechanism employed to track the plan. The overall chaotic and blackbox like behavior of a HRP plan is compensated in some degree, limiting it only to the execution phase. There is also the issue of separating processes that can be executed automatically (like object searches and cleanup) providing an "easier to design" concept.

A complete state diagram with transitions is depicted in [Picture 11]. The plan starts in the *Initialization phase* until it reaches the *finish state*.

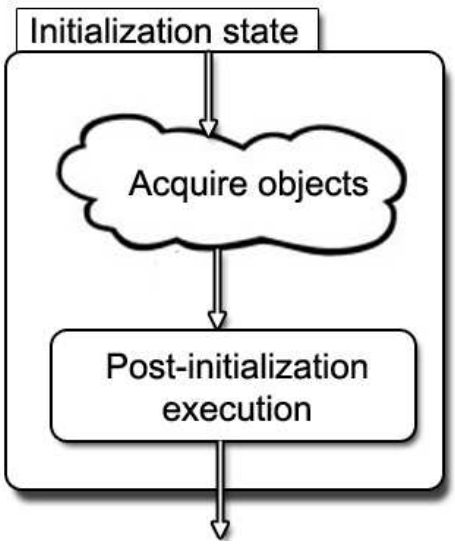


Picture 11: Reactive plan state diagram

6.6.1 Initialization phase

In an average reactive plan, a set of *initiatory* rules are executed, devoted to searching and acquiring objects, the plan needs to successfully perform its task. When *"watering the garden"* plan is executed, the *v-agent* goes on a search for a v-watering can or a *v-hose* or any other object that is able to provide the functionality of watering. These objects have to be specified explicitly in the reactive plans rules, inducing the limits described in (5.2.2) and (5.3.3).

The basic idea of the *initialization phase* to perform all necessary actions (searching and acquiring of objects) to ensure the *execution phase* can be entered in a consistent state. In [Chapter 4] we addressed this phase as "preparation".



Picture 12: Initialization phase

As seen in [Picture 12] the initialization phase consists of two sub-phases, where the *acquire objects* sub-phase is dedicated to search and gathering of objects for the plan and the *post-initialization execution* sub-phase is devoted to executing a series of actions before the plan moves on into the *execution phase*. Post-initialization execution can be used to enter a specific predefined internal state to spare some rules in the execution phase. Designer could use the post-initialization execution to put objects into v-actors hands in a predefined order[1].

To better express the *means (objects)* used by a plan, we need to enrich the reactive plan with an object set and an object-class set. These two structures represent the *plans* requirements for objects, or types of objects. If these requirements are not met, the plan fails. Their semantics were explained in (6.4) and (6.5).

6.6.2 Discussion

Separating the initialization phase allows the developer and designer to better specify and control the process of plans preparation. Also allowing to detect plans that are "doomed to fail" even before the execution starts (the agent may know, that an required object is not available).

Dedicating a separate object[2] to the initialization phase, can lead to a more modular design of an animat mind, providing a possibility for agents to behave differently during

[1] V-*shield* into left hand and V-*sword* into right hand

[2] In terms of a programming language like C++ or Java

the initialization phase, employing the same execution phase. A separate structure provides a vehicle to parameterize the initialization much easier – introducing emotions (anger, fear), internal states (being in a hurry) or external context (night/day), providing a more adaptable behavior without the need to put more complexity into rules of the execution phase.

The issue of a consistent state allows for the execution phase to concentrate on the goal at hand, based upon the assumption that everything is ready to go. The initialization phase can be used as a *fallback position* for correcting a *corrupted execution phase*[1]. This leads to more robust *plan* design and *self-correcting behavior*. The check mechanism for state consistency (all required objects present) shouldn't be omitted when resumed from suspended state into execution.

The [Algorithm 7] can be used as a template algorithm for an initialization phase.

```
4.  update object-set from inventory
5.  update object class set from inventory
6.  update timeouts
7.  if( at least one object list is 'satisfied' && all classes are 'satisfied' )
end(success);
8.  if( all object lists are 'failed' && classes have 'failed' ) end(fail);
9.  tag closest object that is 'present' in a object list or has a class in
the class-list
10.  update object-set with the objects in the surrounding
11.  update object class set with the objects in the surrounding
12.  if ( not next to the target object )
(9.1)   move to the tagged object
○        else collect tagged object
```

Algorithm 7: Initialization phase step

The asymptotical complexity of [Algorithm 7] is approximately

$$\mathbf{O(n*k)}$$

1.'k' is the amount of classes and object in the sets

2.'n' is the amount of objects in the surroundings.

The 'k' parameter can be considered constant, rendering the complexity **O(n)**.

Updating the *object set* and the *object class set* is a simple algorithm of walking a list that can be combined with updating the timeouts.

The steps (7) and (8) are used to mark the entries in the *object-set* and *object class set* with the "*found*" flag, collecting them even when their timeout expires.

There are four distinct states of a object list entry – *present*, *not-present*, *failed* and *found.*

- *present* - indicates that the specified entry (class or object-id) has been collected (and present in the inventory),
- *not-present* – the opposite of *present*
- *failed* – indicates that the timeout has expired and the entry was not found
- *found* – overrides the *failed* state providing a hint, that the object has been sighted and is therefore due to be collected.

Steps (1), (2) and (3) are to update these states of the respective entries.

[1] Another plan drops the object needed by a suspended plan, therefore corrupting its execution context.

Steps (4) and (5) are terminating conditions, *signaling* that the initialization phase has ended and the *execution* preconditions are met.

The step (10) indicates that either the action of collecting an item or moving towards the closest one is to be executed, following the logic of reactive planning – one action per cycle.

6.6.3 Conclusion

Introducing an initialization phase was a natural step. Creating a separated structure allows the designer to choose various approaches that better suit the agent or the conditions. Having an explicit enumeration of item preconditions provided us with the necessary information to be able to compute and decide optimal routes for collecting objects. The in-forward known requirements allow us to mediate the requirements between plans allowing to execute a coordinated object collection (5.3.3), satisfying various plans in a more effective way.

On other hand, the object lists can provide a control mechanism for a expected execution context, when entering the execution phase from a suspended or other phases, providing a fallback point. This feature was natural for the HRP plans, because the high rules dedicated to collecting objects kept the execution context consistent. On the other hand, this approach is not adaptive. The initialization phase could behave depending on how much the context was corrupted, either trying to correct it or failing the plan[1].

In conclusion, separating the initialization phase from the rest of the execution was an obvious step towards more believable behavior without creating a computational complex structure that could represent a considerable bottleneck.

[1] When the amount of missing objects is above 50% it might be better to abort the plan

6.6.4 Execution phase

The actual execution – reaching the goals by applying actions to *v-objects* modifying the *v-world* - is the essence of the *execution phase.* The concept of HRP is represented in SF-HRP in the execution phase, exploiting the HRP concept with minor modifications.

In the execution phase *releaser-action pairs* (4.6) ordered by priority are chosen by ASM[1], where the highest priority *action* with the holding *releaser* is considered a candidate for the execution. The candidate can replace the current executing rule, if all preconditions are met, considering interrupt-safeness and priority. The *action* is a execution vehicle, representing various execution primitives - from single atomic actions, explicit actions (fail, success etc.) to more complex structures like round robin action sets, reactive plan links, signaling states (corrupted-state-detection) etc. The extended actions are discussed in (6.9).

The basic idea of action selection presented in [Algorithm 1] is kept, choosing the highest priority rule with a holding releaser. When found, executing the rules execution vehicle, giving control to it. The execution vehicle can either execute an atomic action, or step down in the hierarchy, or perform any other assigned execution logic (reporting a explicit state – success/fail etc.).

A modified action selection algorithm is presented in [Algorithm 9]. The algorithm is executed in the context of a reactive plan – performing a recursive walk downward the *be-tree* structure until it reaches an rule that can be executed with a result, not leading further deep into the structure.

There are two new structures added to the *reactive plan structure* – the *suspended list* and the *sticky list*.

The sticky list contains by priority ordered sticky rules (3.3). When an active rule candidate is found, it is first check for the sticky flag. A candidate rule with the sticky flag set is inserted into the sticky list. The rules in the sticky list are considered when looking for a new candidate for the position of executed rule.

The suspended list contains rules that have been executed but are suspended (6.6.9). There is also the *executed rule* – the rule in execution – either having the highest priority and a holding releaser, or being interrupt or trigger safe. The executed rule can be either surpassed by a higher priority rule, consequently suspended or its releaser could fail resulting in a fail. There is a possibility the rule ignores the releaser fail, and wishes to be suspended. If the rule is interrupt-safe it can't be surpassed by any other rule, high priority or not.

The candidates to surpass a rule are chosen from the sticky list, the preactive rules and the suspended rules. Different approaches can be used to choose the candidate. Later in the text, we address this issue in detail, proposing bonus functions based on various factor (6.7.1.6).

It is noteworthy, that a *switch* of a rule from executing into suspended can take some time, therefore, it could happen, that the rule is suspended (switched) but there is no one to surpass it. In this situation, the rule is resumed and the execution continues. In the worst possible situation, this could result in a livelock[2] situation. This can be detected by storing

[1] Later in the text (6.7), we introduce an Extended version of ASM.

[2] Similar to deadlock – e.g., when two people meet in a narrow corridor and each tries to be polite by moving aside to let the other pass. They end up in a livelock, not making any progress, mirroring each other's attempt.

the surpassing candidate and when the situation repeats itself, the problematic candidate is excluded from the next candidate choice.

```
Plan Action Selection:

if ( actual rule is interrupt safe ) execute actual rule;

can = get best candidate for execution ( consider sticky and
suspended )

if ( executed rule is suspended ) and no rule with higher priority
was found  then  resume executed rule

if ( executed rule is suspended )
     executed rule = can
     execute executed rule
     return

if ( actual rule is not releaser safe ) and (its releaser is false)
     if ( releaser-fail( executed rule ) = do switch )
        switch executed rule → can
        return;
     elseif ( releaser-fail( executed rule ) == do fail )
          fail executed rule
          return
   elseif ( releaser-fail( executed rule ) = do ignore ) continue

if ( can has lower priority then executed rule )
     if ( can is sticky ) reset sticky timeout( can )
   execute executed rule
   return;

if ( can has higher priority then actual rule )
     switch executed rule → can

execute executed rule
```

Algorithm 9: In-Reactive Plan action selection

6.6.4.1 Tracking location

As stated in (5.3.5.1), the execution phase can be used to track location of execution of the plan providing more information for the plan chaining issue (5.3.5.1). The separated execution phase specifies, where the real execution happens and is tracked, excluding the searches and cleanups from the created statistics.

6.6.5 Conclusion

The execution phase is used to house the real execution body of a reactive plan. The overall asymptotical complexity of RIP is equal to HRP's ASM (3.5)

$$\mathbf{O(n*v*a)}$$

- 'n' being average amount of releasers in one level of the *be-tree*
- 'v' being the height of the *be-tree*.
- 'a' being is the average time to evaluate a releaser. This can be considered a constant.

It is noteworthy, that the resulting asymptotical complexity can change due to the “candidate choice” function, which should be kept simple and optimized, not to create a bottleneck in the RIP algorithm.

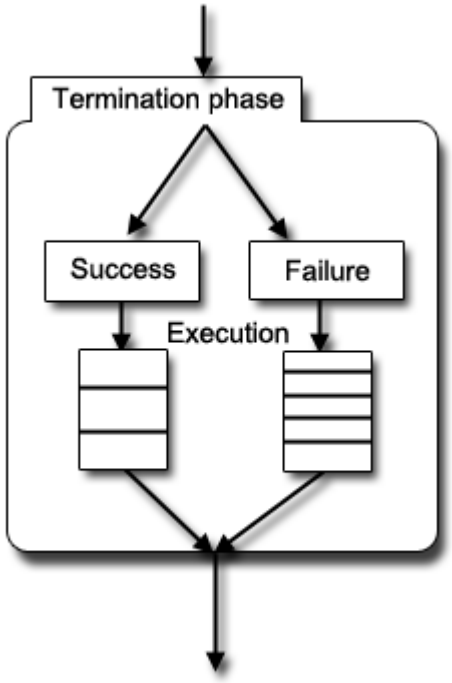
6.6.6 Terminate (Exit) phase

The idea of *terminating phase* is to distinguish the process of *ending (terminating)* the plan from the *execution state.* A separated *termination phase* can be used to invoke specific behaviors based upon the *fail* or *success* [Picture 13]. We have addressed the issue in (5.5).

The *termination phase* is entered, when the plan ends its execution - the termination event occurs. It can originate from the plan itself, when no rule can be chosen active, or created by an executed rule. The parent plan can force a child plan to fail, either when its releaser isn't valid anymore, or the parent has failed or succeeded.

The approach to separate the *execution* and *terminating phase* is based on the observation of humans presenting different behaviors when their activity results in a *fail* or a *success.* When humans *fail*, they tend get angry, frustrated or even depressed. On the contrary, when successful, humans happen to cheer, be motivated or feel joy.

There is also the factor of the *extent* of the *success/fail*. This can be adopted using a floating-point range instead of a binary approach (0 means *fail*, 1 means *success*)



Picture 13: Termination phase diagram

The architecture presented in [Picture 13] illustrates the workflow of a termination phase. The set of actions (an execution vehicle) is chosen based upon the *fail/success* result. When floating-point *results* are used, the action sets can be mapped onto ranges - from *devastating fail* (= 0) up to *tremendous success* (= 1).

The steps of the specified behaviors are executed according to the one-action-per-cycle logic. After the termination phase end, the finish phase (6.6.8) is entered.

6.6.7 Conclusion

The *termination phase* can be used to compensate for constrain of a *behavioral aftermath* (4.6). Providing a place, where to specify and execute possible execution *aftermath* results. Devoting a specified (programming language) object/structure to this task can lead to a more adaptable and modular architecture.

6.6.8 Finishing phase

The *Finishing phase* is considered a specific phase. After the termination phase is finished, the *plan* enters the *finishing phase,* indicating to the ASM or parent plan that the execution and termination has finished and the plan stopped executing.

A *cleanup flag* can accompany the phase, indicating that a *cleanup phase* should be considered, since some objects in the *plans* responsibility should be taken care of. If the *cleanup flag* is not present, the plan has nothing to take care of. Either the parent excluded all its objects, or the plan was aborted, or was marked by designer, not to perform any cleanup at all, making the parent plan responsible for the cleanup.

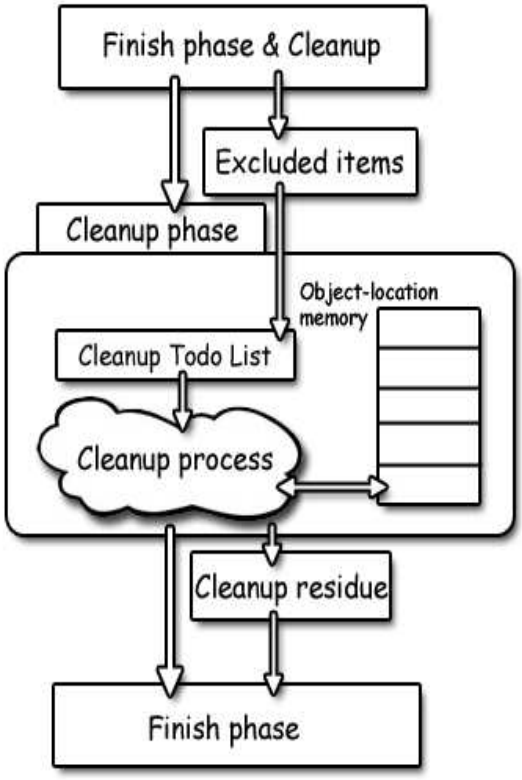
6.6.8.1 Conclusion

Using the *finishing phase* on one hand like a signal for the parent structure (reactive plan, ASM) as the other as a carrier for the *cleanup flag* to indicate a *cleanup phase* was a natural conclusion of the overall concept – everything has to end somehow.

In conclusion, adding a phase dedicated to mark the “end” contributes to a more adaptable design. It could be used to engage specialized engine related execution, freeing of engine resources acquired by the plan during execution, etc.

6.6.9 Cleanup phase

The *cleanup behavior* is considered an important trait of human behavior (5.5). In HRP, this can be overcome by adding rules dedicated to *cleaning up* before the plan ends. This approach can possibly fail, when the *plan* fails due to non-explicit circumstances (external forced fail etc.). We introduce a phase devoted to the *cleanup behavior* – the *cleanup phase.*



Picture 14: Cleanup phase workflow

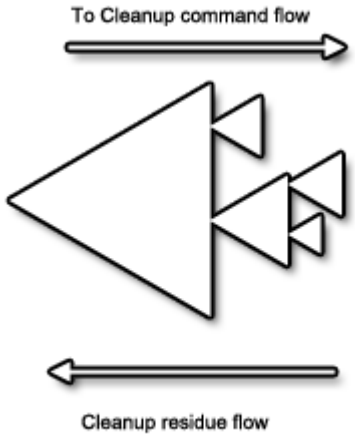
The task of the *cleanup phase* is to be in charge of putting objects back to their respective origins or otherwise take care of them – maybe dropping them.

The idea behind the *cleanup phase* is to be the opposite to the initialization phase (6.6.1), returning the collected objects. The important trait of this phase is, that on the contrary to the initialization phase, where we possibly acquired more object then necessary, in this phase, we might need to exclude various objects. The reason for excluding objects is because they might be needed by different plans, either suspended, chained (5.3.5) or the parent or sibling plans in the *be-tree.* The simple requirement not to disrupt others context is the idea behind excluding items from the cleanup phase.

When the cleanup phase is done, the plan drops the cleanup flag and renders the plan definitely finished and inert. Consequently, it can be removed from execution by its parent.

It is noteworthy, that the cleanup phase might not be able or willing to cleanup all the objects the plan is responsible for (and were not excluded), therefore making it

essential, that after the plan is deemed (absolutely) finished, the *residue objects* are collected – a *garbage collection*. The residue is also collected from plans, which are suspended and therefore not able to perform any cleanup when aborted.



Picture 15: Cleanup flow

Simply put - the parent plan is responsible for its children mess [Picture 15].

6.6.9.1 Conclusion

The cleanup phase can be used to provide compensation for the *pure* form of transitional behavior limitation – *cleanup behavior* (5.5.) - where the actual cleaning can be performed in a separate phase providing a detached and controlled entity. The main advantage is that the phase can be parameterized for excluding certain objects.

In HRP, we can observe that a failed plan won't cleanup the objects that it has acquired - when failing during execution and the cleanup rules are not considered anymore. Also when failing during execution of the rules dedicated to cleanup, the objects are not taken care of. The "cleanup residue flow" provides a way to pass the burden of *cleanup* on to the parent plan.

A plan marked as "no cleanup" (by the designer), isn't performing any cleanup operations when the *finish phase* is reached. This can be used for short sub-plans that tend to be executed regularly, to keep them from walking back and forth and acquiring the same object all over again.

It is noteworthy, that the ASM has to collect and take care of the cleanup residue of the top-level plans. We proposed to introduce a specialized plan dedicated to perform the top-level cleanup (5.5.1).

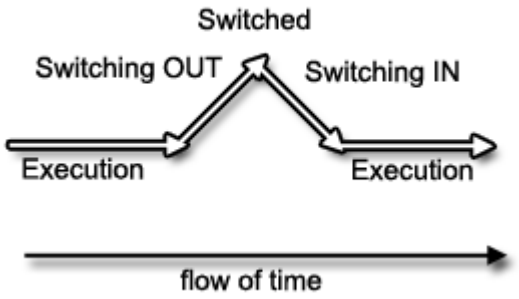
6.6.10 Switching

The idea of switching is based on transitional behavior (5.6), where a specific in-between behavior is needed for a behavior to be discontinued and another behavior to be either started or resumed. In most cases, this happens, when a higher priority behavior suspends a lower priority behavior.

The main concern of *switching* is to achieve a smoother transition from one to another behavior, presenting a consistent state for the suspended plan to resume from.

Switching involves the suspended plan and the suspending plan, creating a pair. The suspended plan enters *switch OUT phase*, reaching the *switched phase.* A plan in switched phase is considered suspended and not able to perform any actions. It can resume its former state trough the *switch IN phase* [Picture 16]. When aborted, the plan enters the *finished phase* directly, performing no *cleanup phase.* It is obvious, not to put suspended plans to execution, when finishing, they are already considered discontinued.

We consider two types of *switching behavior – default* and *related* switching. Default switching is a transition into a *"shared" state.* The shared state is a form of state that all plans can resume from – e.g., empty hands. This state is important in creating a "switch point" that can be used in recovering from specific or emergency situations, providing more robust model.



Picture 16: Single Plan Switching

The *related switching behavior* is intended for plans, which can have a predefined switching behavior pair – the suspended plan leaving a *related state,* where the suspending plan can better catch up.

To provide a picture to the problem, lets imagine a v-warrior, who needs to build a small fort. He has his axe and shield in hands, which served him good during the battle, but now he wants to "switch" from *fighting* to *lumberjack* to get the material for the fort. When employing default switching behavior, he would first clear his hands and then take his axe back into one of his hands. That doesn't look smooth at all, but when employing related switching behavior, he only puts his shield on his back leaving the axe in his hand, "knowing" he will need it. The *switching OUT* behavior was performed by the *fighting* plan in respect to the *lumberjack* plan, providing it with an already prepared situation.

The related switching behavior provides specified behaviors for predefined pairs of plans, making their transitional (switching) smoother.

6.6.10.1 Switch OUT phase

The *switching OUT* is the phase, when the plan leaves his *actual phase* and enters the *switched phase.* The switch OUT behavior should bring the plan into a consistent

state, either in a *default* type of behavior (e.g., putting the used items into the backpack) or *related type* (e.g., putting all items except the axe into the backpack).

A plan in the switched phase is considered suspended and not being able to execute any actions. When a plan containing an executing plan is switched OUT, the branch beneath it has to enter the switched phase. After that, the plan can perform a switch OUT. This can be viewed as a bottom-up approach.

6.6.10.2 Switch IN phase

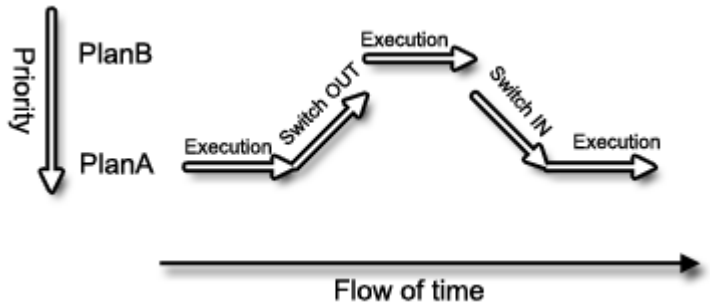
The *switch IN* phase is the opposite of the *switch OUT phase*, the order of suspended-suspending reversed. The switch IN phase occurs either when a plan is finished and a suspended plan becomes switched IN (resumes execution) or a plan is suspended (related switch OUT) and another suspended plan is resumed (related switch IN) and put into execution.

The switch IN phase can be invoked when an executing plan entered his switched phase, but after that, no plan was available to replace him. This could happen when the suspending plan became inactive (its releaser cased to hold) during the switch out phase of the executing plan.

When a plan switches IN, a reverse order is taken, when first the parent node switches IN followed by its executing child. It can be view as top-down, where the switch OUT is view as bottom-up.

6.6.10.3 Switched phase

When a plan enters the *switched phase,* it is considered suspended and no execution can be performed by it. It is a consistent from the point of view of the plan.



Picture 17: Two-Plan Switching

6.6.10.4 Urgency of switching

The issue of *urgency* was shortly addressed in (5.6). The underlying idea is to provide *switching* with a dimension of *urgency,* giving the switching OUT behavior more options how to go into being suspended. An agent could behave differently in various situations concerning the switch of two plans – a agent watches a road and a person comes by, he switches out from guarding behavior differently, when the person is armed or not, providing a better tunable and believable behavior modeling tool.

6.6.10.5 Discussion

Introducing switching into the SF-HRP provides compensation for the *transitional behavior* (5.4) constrain, where the switch OUT can provide the desired behavior, leading to more believable behavior. Specifying a default IN/OUT behavior and related IN/OUT behaviors can lead to more smooth execution.

With switching introduced into the concept, some new problems arise that need to be addressed – *misguided switching* and *finish switching.*

The misguided switching appears, when the plan (A) is performing related switching OUT, with regard to a specific plan (B). But the switching can take more then few cycles to complete, and the circumstances could change so that a plan (C) has a higher priority then the plan (B), which was used as a *switching reference.* Therefore, when (A) finishes the switching, plan (C) is now there to start its execution with a context ready for plan (B).

In our opinion, there are three ways how to solve this

1) A simple detection mechanism is put into the ASM, where after the plan has finished switching, the active plan is checked to be the same like the one the switching plan switched to. When a corruption is detected, the switching plan is asked to perform the default switching behavior to ensure a shared state.
2) The above proposed detection mechanism performs a related switch IN on the switched plan and retries the related switch OUT of switching plan regarding the new detected target (the C plan)
3) To enrich the related *switch OUT behaviors* with a complementary behavior, which can be used to move from the related switch OUT behavior into the shared state.

The issue of switching IN is trickier then the switching OUT, because the lower priority plan that was switched OUT can be switched IN either when the higher priority has *finished* or was *suspended*. When suspended, the concept works, the switching OUT plan provides a state for the switching IN to perform as planned.

The finishing phase provides an obstacle, because the finishing plan might perform cleanup or otherwise render the overall situation not available to be switched IN (in a related manor).

Simple scenario considering the v-warrior from earlier coming home – he puts his shield on the wall, but to do that, he needs to put his axe on his back. He excludes the axe from being “cleaned up”, but he won’t put it back into his hand, because he finished his cleanup phase and is done with the plan.

There are multiple possibilities how to solve this issue

1) not using the related switch IN type of behavior, considering the context always turn to default consistent state – this can provide undesirable behavior
2) introducing the “related” concept to the termination and cleanup phase, when certain actions will be performed in respect to the successor plan that switches IN.
3) introducing the “related” concept to a expansion of the finish phase, where a specific related behavior would be executed after the cleanup phase ends. This phase should also be executed when no cleanup is performed.
4) introducing a consistency check for the related switch IN phase, where certain conditions have to be met or the default switch in behavior will be executed.
5) When a plan is finished, the default switch IN is used for the resuming plan, expecting the finishing plan to leave the agent in a shared state. It could be good to consider creating robust default switch IN behaviors, to be able to compensate for glitches in the shared state (a object was left in hand etc.)

It is noteworthy, that when employing the *related concept* to the termination, cleanup or finish state, a problem similar to *misguided switching OUT*, where the target for the related termination/cleanup/finish switch IN changes during the execution of the switching behavior. The solutions for this are the same, because it is the same problem all over again.

6.6.10.6 Conclusion

The *switching behavior* concept provides a suitable solution for the limitations of HRP presented in (5.4).

For the problems presented in the discussion above, we propose the usage of (3) for the misguided related switch IN/OUT, and (5) for the *finish switching issue*. They provide the in our opinion best approach to cope with the issues at hand.

6.6.11 Emergency phase

The *emergency phase* is a special phase that can be entered at any time, providing room for specialized execution. When an emergency occurs, ASM forces the active top-level goal to enter the *Emergency phase.* After the execution of the emergency behavior finishes, the plan resumes his former state.

This phase can be used to model various “short reflex like behaviors” - gives the designer a specific phase that can be entered at any time.

6.7 Extended Action Selection Method (EASM)

To clarify things, we have to introduce the *Extended Action Selection Method* (EASM). It is similar to the ASM presented in [Algorithm 1]. Consists of a *top-level phase* (TLP) and a *recursive in-tree phase* (RIP). The TLP phase is engaged on the top of the *be-tree*, where the reactive plans are linked with releasers to the top-level goal ("be alive") [Picture 8].

The RIP algorithm conceptually similar to the recursive algorithm shown in [Algorithm 1] – stepping down the hierarchy of a *be-tree* until it reaches an atomic action that can be executed.

The TLP's main task is to manage the plans at the *top level* of the hierarchy, and invoking the RIP algorithm on the chosen *active plan*.

6.7.1 TLP (Top Level Phase)

TLP get execution candidate ***(executing):***

//takes the items with a holding releaser
suspended-candidates = valid-suspended-list()
//takes the highest priority
sticky-candidates = valid-sticky-list()
//takes the holding releasers, not disabled, highest priority
preactive-candidates = valid-preactive-list()

//changes weight and priority
apply-chaining(***executing***) to ***suspended*-candidates**
apply-chaining(***executing***) to **sticky-candidates**
apply-chaining(***executing***) to **preactive-candidates**

//filters the top priorities
filter-top-priority(***suspended*-candidates** *)*
filter-top-priority(**sticky-candidates** *)*
filter-top-priority(**preactive-candidates** *)*

//chosen by weight
suspended-lead = *choose_top(* **suspended-candidates** *)*
sticky-lead = *choose_top(* **sticky-candidates** *)*
preactive-lead = *choose_top(* **preactive-candidates** *)*

return highest priority(**suspended-lead**, **sticky-lead**,**preactive-lead** *)*

Algorithm 12: get execution candidate

```
TLP check releaser:

if ( executing is releaser-safe ) return

candidate = get execution candidate

if ( check releaser( executing ) == IGNORE ) return;

if ( check releaser( executing ) == SWITCH )
   executing switch out( candidate )

if ( check releaser( executing ) == FAIL )
   atExit( executing )
   executing do fail

if ( check releaser( executing ) == HOLDING //is true
   executing perform releaser check on sub-tree
```

Algorithm 13: TLP releaser check

```
AtExit ( p ):
create exclude list
for every plan in suspended plans
   exclude list + collected items of  plan
//the in initialization don't have all collected
   if ( plan in initialization phase )
      exclude list + need items of  plan

for every sticky plan in sticky stack
   exclude list + need items plan
for every preactive plan
   exclude list + need items in plan

exclude items ( exclude list ) from p
```

Algorithm 14: excluding items from plan

```
TLP pre-exec:

candidate = get execution candidate( executing )

if ( executing is interrupt safe )
    if ( candidate is sticky ) add to sticky( candidate )
    return

if ( executing is finished && cleanup is done )
   collect cleanup residue
   if ( candidate is suspended )
      candidate switch in ( executing )
      remove candidate from suspended
   else
      if ( candidate was sticky )
         remove candidate from sticky list

   executing = candidate
   return

if ( executing is switched )
   if  ( candidate is none )
      executing switch in
   else
      executing is suspended
      executing = candidate
      if ( candidate was suspended )
           switch in (candidate)
           remove candidate from suspended
      else
           if ( candidate was sticky )
              remove candidate from sticky list
   return

if ( executing has lower priority candidate )
   executing switch out ( candidate )
   if ( executing is switched )  executing = candidate

if ( candidate was sticky )
    remove candidate from sticky list
if ( candidate was suspended )
    remove candidate from suspended list
```

Algorithm15: TLP pre-execution

```
TLP exec:

check releaser( executing )
pre-exec()
result = executing – RIP
if ( result is CONTINUE ) return
if ( result is FAIL )
   atExit( executing )
   executing do fail;
if ( result is SUCCESS )
   atExit( executing )
   executing do success;

return;
```

Algorithm16: TLP execution

The TLP algorithm [Algorithm 16] first checks the *executing* plan for a holding releaser. A failed releaser can result in ignoring, or requesting a *fail* or *switch (suspended).* When failing, the plan receives a list of objects it should exclude from its cleanup phase - the objects originate from suspended, sticky and preactive plans either being already collected or to be collected as needed objects. This approach saves time and keeps the contexts of other plans uncorrupted, possibly keeping object for other plans. This concerns the issue of *cross-plan preparation* (5.3.1).

When the plan passes the releaser check, either by a holding releaser or a having a *releaser-safe* flag, a best suitable candidate to replace it is searched. The *interrupt-safe* flag can overrule this, keeping the *executing plan* from being surpassed. The candidate is chosen from the preactive, suspended and sticky plans. If a suitable candidate is found and its priority is higher then the priority of the *executing* rule, the *executing* is switched out.

The interesting part is the choosing of candidates, where first all possible candidates are collected, their releaser holding, or being the highest priority among the sticky rules. Then the *chaining* process is applied, possibly modifying the weights and priorities of the candidates. Then the highest priorities are sorted out, possibly applying the *weight* ratio again for equal priorities. After for every set – the suspended, preactive and sticky – a lead candidate is found, the best of them is chosen. When equal, the *suspended* are preferred, followed by the sticky and the last considered are the preactive rules. The suspended, preactive and sticky sets are don't intersect.

A noteworthy trait to introduce into the algorithm is to keep the changes induced by chaining until the plan is finished, keeping the chaining intact, not allowing others to surpass the current candidate benefited from chaining in the next iteration.

The asymptotical complexity is

$$\mathbf{O(n*a)}$$

- 'n' denotes the amount of plans in the top level of the *be-tree*
- 'a' denotes the time required for a releaser check

  the sticky, preactive or suspended list contain less items then the total amount of plans on the top-level.

6.7.2 RIP (Recursive In-Plan)

The RIP algorithm is similar to TLP with respect to the changed structure of SF-HRP. It consists of a *pre-execution phase* [Algorithm 17] and an *execution phase* [Algorithm 9], which is engaged if the plan is in execution phase (6.6.4). The validating of sticky rule timeouts is done in the pre-execution phase.

```
Pre-execution:

check local phase // see Algorithm 18

for every sticky list  entry
     if ( timeout expired( entry ) ) remove  entry

if ( plan state is execute )
   Plan Action Selection //see Algorithm 9
```

Algorithm 17: RIP pre-execution

The first what the RIP algorithm does is to check the *local state* of the executed plan [Algorithm 18]. If the state is execution, the [Algorithm 9] is executed resulting in

performing the RIP algorithm in the context of the chosen action. An action in the leaf of the *be-tree* is executed as a overall result.

check local phase:
(0) if (state is switched) return
(1) if (state is *finished*)
(1.1) if (state is *cleanup*) do_cleanup()
(1.2) return;
(2) if (state is *emergency*) do_emergency()
(3) if (state is *terminate*) and (interrupt-safe == false)
(3.1) if (state is *success*) do_success(); return
(3.2) if (state is *failure*) do_fail(); return
(4) if (state is *switching in/out*) perform switch in/out; return
(5) if (state is *inititialzation phase*) perform initialization step; return;
(6) if (state is *terminate*) and (interrupt-safe == false)
(6.1) if (state is *success*) do_success(); return
(6.2) if (state is *failure*) do_fail(); return

Algorithm 18: checking local state

The idea behind [Algorithm 18] is to execute specific routines based upon the actual phase of the plan. It is important to understand, that a plan in the executing branch of the be-tree has to propagate its phase downwards first. E.g., when the plan enters a switch-in phase, it is spread this to its executing rule. The behavior of the parent can be engaged (executed) only after the executing child has reached the phase. We call this the *phase propagation.* Without it, the be-tree would end up in an inconsistent overall state.

The opposite direction of communication (from child to parent) can be managed trough return values of functions and thrown exceptions[1]. The executing rule should report a *fail*, *success* or *no-result* to the parent structure. The parent structure then invokes a command to either fail, succeed and returns the control to its parent by returning a *no-result*. This is important, to keep the fails/successes contained to the current level. When a rule fails/succeeds, it doesn't indicate its parent plan fails.

The next issue is the employment of exceptions generated by specific actions (7.3) and propagated fails and success (4.4), which can be introduced as an exception, which is caught, altered and possibly rethrown.

These implementation specific characteristics might not be available for every programming language, but illustrate the intended behavior or back propagation of results up the *be-tree* when execution calls unwind after completed.

6.7.3 Conclusion

The TLP phase of the EASM is responsible for managing the top level of the *top-level goal* of the *be-tree*, where the RIP is the actual recursive algorithm that walks the *be-tree* structure resulting in an execution of an atomic action.

[1] Considering a procedural language like C/C++ or Java

6.8 Extended Actions

The action is the part of the rule, that is executed when the rule's releaser holds and it has high enough priority to be executed, Denoted as *exec i*n *reactive rule formalism* (3.3)(4.6). The HRP model provides only action sequences and single atomic actions. A link deeper into the *be-tree* structure – a reactive plan link – can be also considered an action.

Inspired by [32], we propose to introduce more complex *actions* into the model

- *cycle actions* – the action sequence is repeated when finished
- *n-cycle actions* – the action perform an preset amount of iterations
- *one shot actions* – the action performs once, and then renders itself disabled until reset
- *n-shot actions* – the action performs a given amount of times and disables itself
- *probabilistic* – a set of actions (not necessary atomic) performed by random choice
- *random once* – like *probabilistic* but a choice is never repeated (until reset)
- *fail proof* – on fail, reset the execution of its sequence
- *random multi-plan* – the action consists of multiple plans, when first executed, a plan is chosen by random from the set and is considered to be a permanent choice (until reset)
- *conditioned multi-plan* – same like behavior like the *random multi-plan* with the exception that the plans are chosen by condition or provided reasoning function
- *goal based choice* – this action consists of a *goal*, when executed, a plan is chosen based upon their ability to satisfy the goal. The plans ability to satisfy a goal can be specified introducing goal metadata to a plan (6.3)
- *multi goal* – the action is a set of goals which all should be satisfied. The plans are chosen in the same manor like in the *goal based choice* action. The action could get parameterized with a retry amounts for specific goals.
- ...

In regard to the proposed changes in the reactive plan structure, we propose to include actions that can explicitly influence state changes. The fail and success actions have already been presented, but it might be of use to use a *toInit* and *ForceSuspend* actions, where the toInit action returns the plan to its initialization phase[1] and the ForceSuspend suspends the current plan for given set of cycles[2].

6.8.1 Conclusion

A modular concept viewing the action of a rule as a separated structure with a known interface, allows introducing various complex actions providing necessary behavior diversity. The interface should contain basic methods – *execute, fail, success, check-releasers, get-state, is-interrupt-safe, is-releaser-safe, get-cleanup-residue* etc. Also a command interface has to be included, containing functions representing various commands related to *state changes, resetting,* etc.

The *goal based choice extended action* provided a suitable compensation for the *process based choice for a goal* presented in (5.2.2.1), where a plan to be executed can be selected based upon the goal intended to be satisfied.

[1] Can be used when a corrupted execution context is detected during execution.
[2] To prevent it from being chosen as a candidate the next cycle

6.9 Per-plan blackboard (memory)

A *blackboard* like concept of per-plan memory can provide a useful communication frame for rules in a plan to share information. Rules (their actions) could access and post information on the blackboard on its own level or higher, making it available to rules on the same or lower level of hierarchy (in a *be-tree*) – structures residing “upwards” are *instantiated*, where the structures *downwards* a blueprints.

This concept can be exploited to pass information between separated executions of a rule – marking objects unavailable, paths blocked etc.

We provided only the outline of the idea for further discussion.

6.10 Conclusion

Introducing *statefullness* to the reactive plan structure gave us much more needed control over the behavior of the reactive plan, being able to track its progress from the outside, staying compliant with the demand of responsiveness of the agent and performing in timely fashion – keeping the base concept of reactive planning intact.

The ability of auditing the plan to some degree by a external system provides us with the opportunity to introduce the EASM with mechanism to provide functionality to overcome the rest of the limitations discussed in [Chapter5], also incorporating the solutions provided in [Chapter 4] into EASM.

We proposed a state full modular approach to reactive plans, where plans are not only containers for reactive rules, but shifted into a more "self-aware" structure, able to handle specific states and to provide better control for their content (reactive rules). The modular approach to phases can provide a more adaptable concept being able to cope with complex demands on behavior, introducing various object designs (a programming language point of view) to accommodate various needs for the certain phases to perform.

The main advantage, besides the statefullness, is the explicitly specified needs for a plan to work – the object requirements. Upon these information sets, various approaches to overcome difficulties can be stacked to provide more believable behavior emerging from analysis of these sets. Also the separation of execution provides a body to inspect, providing more accurate statistics – location, duration of execution etc. The statistics will be more precise excluding contamination from the search, collect and cleanup behaviors, executed at the start and end of average reactive plan. The concepts are all backwards compatible with the HRP, providing the same functionality, when rendered unavailable, providing only the execution phase.

Providing specified phases dedicated for terminating a plan allows us to examine the results and influence other components of action selection or the agent's brain. Switching behavior allows us to transit between behaviors in a better and controlled fashion.

The SF-HRP can always be used in a "HRP mode" and therefore it can be considered equal. The only limitation we didn't cover by this chapter is the *adding of intentions* (5.2.1), which will be discussed in the following chapter.

It is noteworthy, that the SF-HRP can be used in a modular fashion, allowing introducing various approaches to different phases, providing a even more adaptable design. With minor changes various character traits can be introduced to the agent to provide even more believable behavior.

6.11 Drawbacks

The main drawback of the SF-HRP is that it provides a too well structured behavior, when employed in a rigid way. Most notably, when plans tend to perform a "wave like" execution pattern – the acquiring, execution and cleanup of objects is done in waves, creating a wave like pattern.

This can be modeled in SF-HRP, either by splitting the "waves" into sub plans and using propagated fails, when one of the sub plans fails, failing the whole endeavor.

7 Modifying the be-tree

The major disadvantage of a *be-tree* is that it is a static structure and HRP provides no techniques to modify it to any extent. The issue of adding intentions (5.2.1) raised the question of modifying, based upon request from an external or internal (regarding *be-tree*) source.

We propose to add a set of special actions – *the modifiers* – which can influence the structure of a *be-tree*, either by requesting to add, remove or modify nodes. To provide more flexibility, we propose to introduce a set of specialized *virtual nodes*, which contains other nodes, posing as any of those nodes, the choice based upon a releaser and priority. To further improve the flexibility of a *be-tree*, we propose to add a set of links - *channels* - which can be used to influence nodes by other nodes, forming a *dependency grid* – e.g., when one node fails, all others fail, or are removed from the *be-tree.*

To compensate for the *adding intentions* (5.2.1) constrain, we propose the Intention-add Action (IaA).

7.1 Intention-Goal-Plan map (IGP map)

Inspired by the Belief-Desire-Intention [12] model, we propose to introduce an *Intention-Goal-Plan map* (IGPmap) to reactive planning. The IGPmap is a database of *intentions* and their related goals with plan alternatives assigned to them. Providing a three-layer structure, from which one entry is presented in [Table 9].

Intention	Goal	Plan alternatives
[20]Have a beautiful garden[10]	[30]Water[1]	[20]Water by watering can
		[60]Water by hose
	[30]Weed[1]	[30]Weed
		[30]Use chemicals
	[10]Take care of trees[3]	[10]Cut down trees by axe
		[90]Cut down trees with a chainsaw

Table 9: Intention-Goal-Plan mapping

In [Table 9], the first column is the *intention*, consisting of three attributes – the *base weight*(“[20]”), the actual *intentions* (“Have a beautiful garden”) and a *base priority* (“[10]”). The attributes of *weight* and *priority* are informational, to provide a hint or default parameters for other mechanism (like IaA).

The second column is a set of goals (“Water, Weed, Take care of trees”) that are to be executed to accomplish the intention. Each goal is provided with a priority (“[1]”) to be used in conjunction with the *base priority* and provides information for the ordering of goals. For goals with equal priorities, a *weight* (“[30]”) is provided, to allow specifying preferences.

The third column contains the plan alternatives to respective to given goals, provided with a *weight* to allow specifying preferences toward various plans.

7.2 Extended IGPmap (EIGPmap)

It is obvious that the IGPmap might not be able to specify a more complex set of requirements on accomplishing an intention. In the IGPmap, the set of goals is a conjunction, where all goals have to be finished successfully to accomplish the given intention. For every goal, there is only a list of alternatives, distinguished by their weight.

The EIGPmap can be a next step to provide a more detailed structure for intention mapping.

The Intentions should be provided with multiple alternatives of goals to represent different approaches the intention can be accomplished - either by gardening work or by hiring someone who can do it for us.

```
"Have a beautiful garden" →
 {P(water),P(weed),S(take care of trees)),(P(hire a professional)}
        Code 7: Extended intention-goal mapping formalism
```

The outline presented in [Code 7] is without the given weights and priorities used in (7.1).

The other issue with *intentions* is that some of the goals are necessary to be satisfied and others are not – *primary* and *secondary* goals. Where *primary* goals are required for an intention to be satisfied, and when they fail, the whole alternative is considered failed. When all alternatives fail, the intention was not accomplished. On the other hand, secondary goals are optional and when they fail, it has no impact on the goal set alternative or overall perception of fail/success of an intention. In [Code 7], the primary goals are enclosed in a "*P(goal)*" where the secondary goals "*S(goal)*". The sets of goals can have a priority and weight provided, either specifying a order of execution desired or preferences for a given set of goals. Also goals in a goal set should have priorities and weights specified, providing ordering and preferences.

The same approach for intention-goal mapping can be applied to *goal-plan* mapping, not only presenting alternatives for a goal, but ordered alternative sets with primary and secondary plans.

```
"go to shop" →
{ ( P(get gas),P(by car) ), (S(get ticket),P(go by bus) ) }
           Code 8: Extended goal-plan mapping formalism
```

In [Code 8] we omit the priorities, which denote ordering and weight. The idea behind the goal-plan mapping is the same like the idea for intention-goal mapping, only applied to goals and plans.

To both, the intention-goal mapping and the goal-plan mapping, additional information like amount of fails or successes (4.4) etc., can be added, to better express the structure.

7.3 Modifiers

The idea behind *modifiers* is simple – provide specialized actions to alter the *be-tree* structure – add, remove or modify nodes – change the tree during runtime of the agent, not being limited to the initial static setup made by the designer. The modifiers don't add levels to the *be-tree* and therefore, they only need a offset[1] upwards from their position in

[1] Utilizing the same concept proposed in (4.4.1)

the *be-tree* specified. The modifier can inflict changes only to the structure along its path to the top-level goal.

When a holding releaser invokes an action, it might change the structure of the *be-tree* - remove a branch in the tree, or add a new one. A possible modification – rising priority, change of weights etc. could be considered. The approach of modifying a branch (the one where the rule resides) could be employed in various ways – either by reinforcing the preferences of the rule based upon successful execution or rising the priority making the rule or a parent structure, more important. This alterations put more stress and complexity on the over heading (E)ASM.

Adding branches can be exploited by external or internal sources (regarding the *be-tree*) creating opportunity for adding intentions into the concept.

Also form of learning could be introduced, where an action could be a *talk to tutor* action enhanced by a *modifier trait.* The action could provide a new set of top-level goals to the *be-tree,* simply by executing. In the same manor a "forgetting" mechanism can be employed.

This approach raises various new questions, which are concerned with the consistency and management of such changes, not to bloat the *be-tree* to unmanageable proportions.

7.4 Channels

It might be necessary to tie some plans in the be-tree together in a *dependency grid* - when something happens with a plan[1], the plans that are dependant on its execution or are related in some other way, need to know. It can be seen as a shared releaser condition, where a plan can check for the results put into a shared variable. We propose to incorporate a *channel-like* concept into reactive plans, providing them with *connector* like interfaces, to be able to propagate necessary information more effectively.

The channels could be used to perform a cascade like failure or disabling of plans, e.g., when a plan destroys an item and other plan depend on it, they are aborted, possibly removed from the *be-tree.*

We only propose this concept; it is beyond the scope of this thesis.

7.5 Virtual nodes

When we look upon the structure of a *be-tree,* we can perceive nodes, where every node has a priority, releaser, weight and other associated structures (*state*, *object lists* etc.).

We propose to add a specific node - *virtual node* - to the *be-tree* structure. It behaves similarly to a normal *reactive plan node* (containing rules).

A virtual node:

- contains nodes that have releasers, priority, weight, like a *reactive plan*
- its releaser is a conjunction of the contained nodes holding releasers
- its priority is the priority of the contained node with the highest priority and a holding releaser
- other attributes are a conjunction or the best possible choice (like weight) among the contained nodes with holding releasers
- the node has no execution or initialization phase of its own
- its switching is instant, when the executing sub node has finished switching
- may contain a termination phase, to act upon fails or successes of contained nodes

[1] fail, success, is removed from the *be-tree,* etc.

- when entered for execution, it behaves like a common reactive plan invoking RIP on the contained nodes.

A virtual node can be called a *"aggregation node"* or *"unification node"*. It is a masquerade for the nodes beneath it.

This concept can be used to give more versatility to the *be-tree* structure, encasing certain plans that might have a common point of failure, when one containing node fails the whole sub-tree with the virtual node is disabled, but providing a structure that can mimic any node or set of nodes (with the same priority) for chaining or other higher level employed concepts.

7.6 Intention-add Action (IaA)

To make use of the concepts presented in this chapter, we propose the *Intention-add Action* (IaA).

The main goal of IaA is to compensate for the constrain of *adding intentions* presented in (5.2.1), providing the *be-tree* structure with more versatile approach making changes to the structure available at runtime.

The IaA employs the concept of modifiers and the can make use of the (E)IGPmap, where in the body of the action, an intention is specified. The result of such specification is a *be-tree* sub-tree structure that is hooked among the top-level behaviors.

In a IaA the structure can be predefined or created on demand based upon the data in the (E)IGPmap. We distinguish to approaches to the creation of a *be-tree* substructure based upon the (E)IGPmap.

1. The goals associated with the intention are considered *top-level* goals, where their priority is computed based upon the *intention priority* base and the *goal priority* in the (E)IGPmap. The same applied to the *weights.* The goal alternatives are put into the reactive plans with equal priority (they are alternatives) with the proper specified *weights.* Their successful execution is considered being a success for the goal.

   In the *IaA* an releaser has to be specified and is associated with every *top-level* goal created from the goals in (E)IGPmap. A *shared condition* considering the fail of any of the goals can be incorporated into the releaser, to provide a fail when any of the goals fails. Also *channel* [4.5] can be used to mimic this.

   To properly cleanup after the execution either succeeds or fail, a *modifier* [7.3] has to be included into the *termination phase* of the *top-level* goal, to request its *removal* from the *be-tree*.

2. A different approach is to create a *virtual node* representing the *intention* and inserting the goal nodes in the same fashion presented in (1). This approach can create a easier to cleanup and fail-propagation setup, where when one goal fails, the whole *virtual node* fail requesting a removal from the *be-tree.*

3. The sub-tree is already constructed in the IaA and it is directly hooked to the top-level

As seen, the IaA in conjunction with other proposed concepts can provide a tool to add intentions to the reactive plans concept. It is noteworthy, to disable the IaA rule after execution or provide a shared condition for its releaser and the inserted intention, to hinder the IaA from repetitive execution.

7.7 Conclusion

In this chapter, we presented concepts that can be used to successfully modify a *be-tree* in various ways, providing a more dynamic but adaptable design. In conclusion, the limitation presented in (5.2) can be compensated by introducing the IaA into the available actions setup. The attempt to insert intentions can also originate from outer sources, providing the system with ability to influence the action selection in a *be-tree* by other components of the agents mind.

The (E)IGPmap can be introduced on a per-agent basis, providing every agent with a own approach to attend to intentions being able to model single behavior traits more believably.

8 Prototype implementation

We developed a simple prototype to present some of the proposed improvements on a live system, providing a reference and presenting our conclusions in action. Some of our improvements may need to be tweaked based upon the application engine they are introduced into.

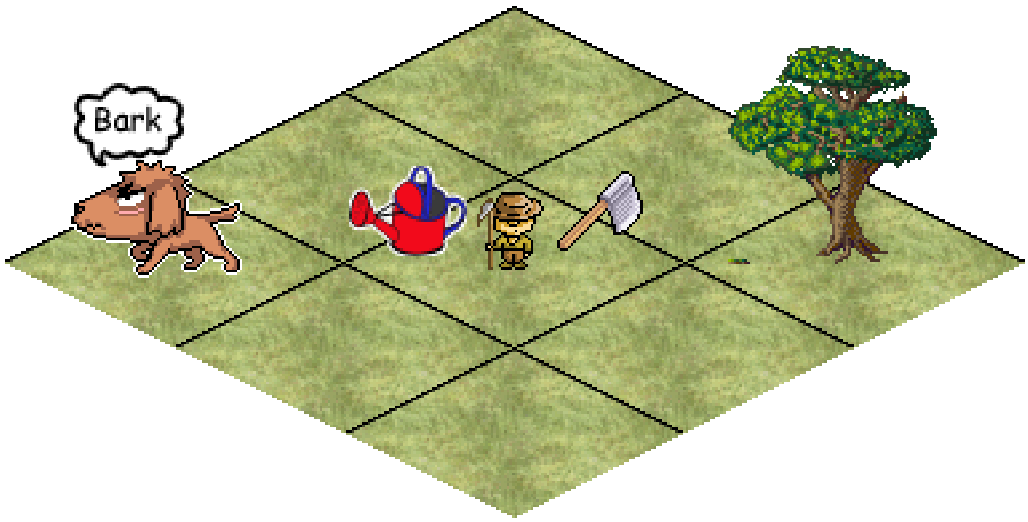
We decided to make use of the C++ language, because of its flexibility and speed. We extensively exploited the concepts of STL and Object-oriented programming in our prototype.

8.1 V-World overview

The agent is situated in a 2D world consisting of equally difficult to overcome tiles presented in an isometric graphics engine. Every tile can be skinned with multiple layers of overlapping images, providing better visualization. Everything, except the world itself, is considered to be an object that can be perceived, collected, used or equipped. It can be equipped with various skins, of which one is active at a moment, representing various traits of an object [Picture 18] – a sharp and blunt axe.

Picture 18: A sharp and blunt axe

The v-world runs in discrete time slices, providing enough time for an agent to calculate its next step based upon the SF-HRP approach. Every agent has a bottomless inventory available, where it can store collected items. A human like agent has two hands (left and right) for wielding objects it has collected. Also hands are used to drop objects, where the object has to be equipped first and then dropped. In [Picture 19] two agents can be seen. A barking v-dog and a v-person holding two item in its hands – a axe in the right hand and watering can in the left hand. A tree can also be seen.



Picture 19: Agent standing next to a tree with objects in hands

Agents are equipped with a “brain” based upon a SF-HRP approach present in this thesis and are presented with short scenarios. The world itself is not interactive. In [Picture 20] a variety of agents can be perceived. On the top are two v-dogs, one of them

cheering and the other being sad or eating. In the front are two v-humans gloving in respect for their actual emotion[1] – red being angry about a failed plan, yellow being happy about a success. There are two objects in the middle – a piece of v-meat and a sharpening stone for the items with a blade.

Picture 20: Agents displaying emotions and objects

8.2 Architecture

Our prototype consists of three main components – the graphical engine, the object engine and the artificial intelligence (AI) engine.

The graphical engine provides the visualization for the world and objects, providing images and image collections. The SDL library is used to perform the rendering.

The object engine provides basic functionality involving the object management, object class management, registering new object classes symbols, registering object in the v-world, etc. We use singletons for manager objects.

The AI engine is responsible for maintaining and providing primitives for the agents mind to employ. The agent makes use of a virtual body [Picture 19], having sensors that can provide information about the surrounding – putting location of object in visual range into object-location memory [Picture 10]. The virtual body has two hands and a bottomless inventory for disposal. The action selection part of the agent's body is regularly (every cycle) tasked with selecting the next action to be performed, making memory and other components of the body available.

8.3 Objects

Objects are a vital part of the v-world everything turns around them. They can be collected, used, stored in an inventory, have various traits and functionalities. We implemented a concept of *intelligent objects*, where an object can identify another object it is used upon, choosing the proper action. E.g., an v-axe used on a v-tree, transforms the

[1] Our prototype doesn't introduce emotions to the HRP model or simulate them in any way. The agents expressions are loosely based upon the actual executing context – e.g., fail/success or eating food, seen something, etc.

tree into wood logs. Objects also carry besides a unique identifier information about their object classes and position in the v-world. From the *ObjectBase* class, all objects in the world are inherited. The *ImgObjectBase* is used for objects that need graphical representation in the v-world.

8.4 Trigger system

The trigger system provides a basis for the evaluation of rules in action selection. The triggers are rooted trees, where nodes are either logical operators (or, and, not) or conditions evaluated true/false creating a *condition tree* [Picture 21].

8.4.1 Poll Conditions

Poll conditions are conditions querying certain information comparing them to the awaited result e.g., "standing next to a tree". We provide a set of *actor related* conditions that query specific information

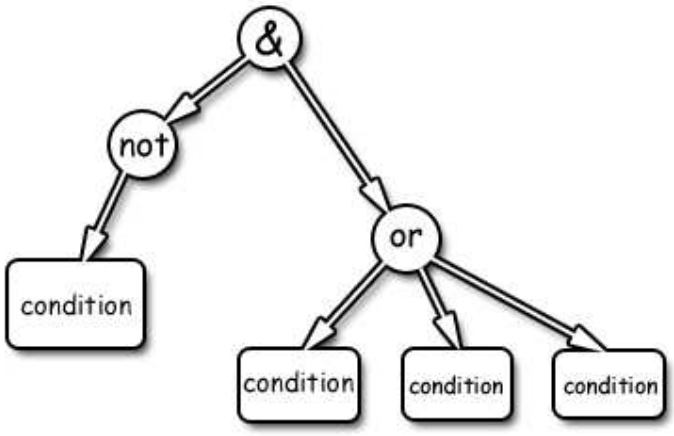
- visual percepts
- nearby objects
- standing next to objects
- checking certain object classes/identifiers holding in hands
- checking object classes/identifiers contained in inventory
- checking actor related properties
- checking holding object properties
- …

8.4.2 Event Conditions

Event conditions are evaluated true, if certain events occur. The events are registered and maintained by a singleton manager object, where an event condition can register at a event line, listening to incoming events. Events are implemented into our prototype, but not used.

8.4.3 Condition tree

The condition tree [Picture 21] is a simple structure, where leafs are conditions and other nodes are boolean operations – and, or, not. The structure allows performing lazy evaluation, providing reasonable performance even on larger formulas.



Picture 21: Condition tree

8.5 AI engine

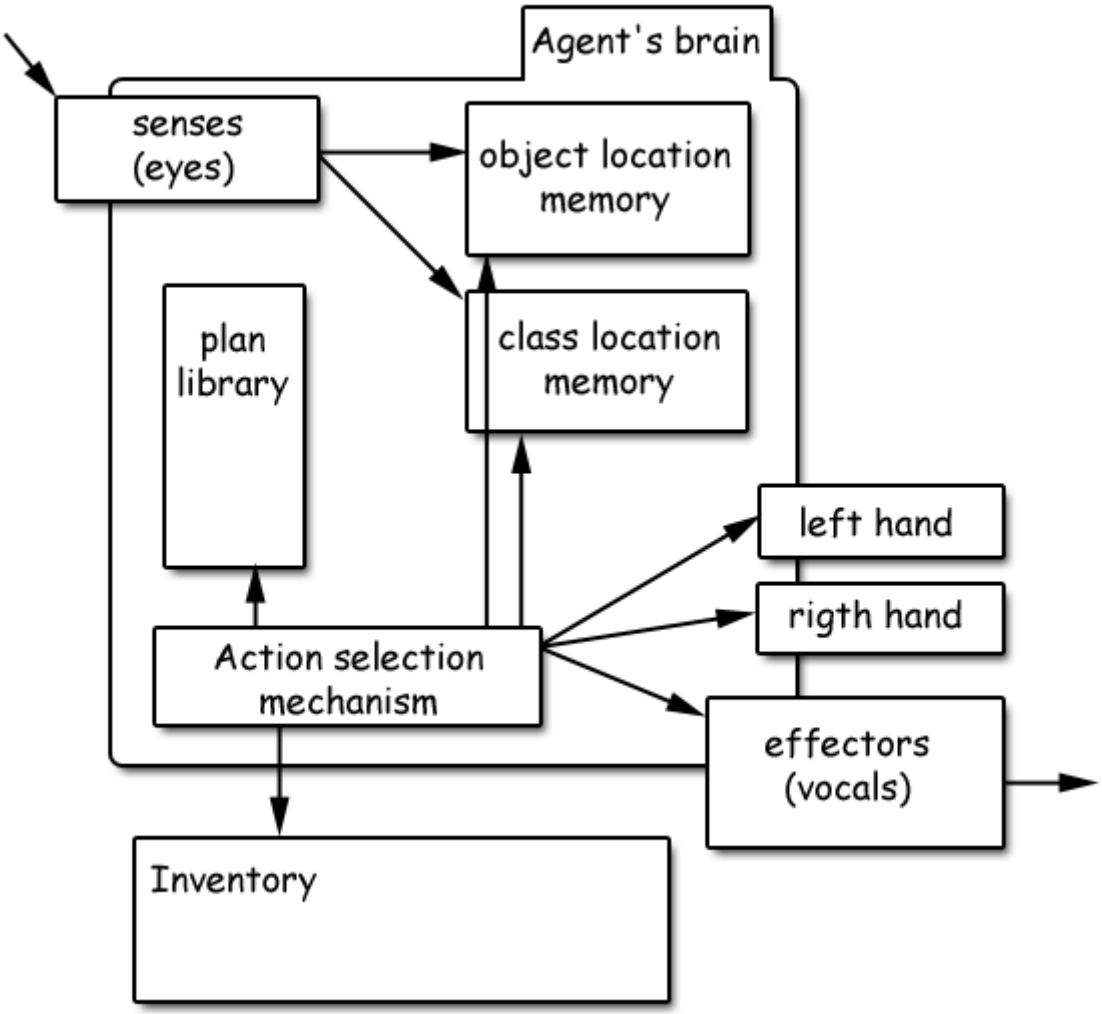
The AI engine is the driving force behind the agent's manifesting behavior. Providing structures like memory banks, plan libraries and action selection auxiliary structures, responsible for managing specific phases of a plan

Initialization, Termination, SwitchIN, SwitchOUT, Emergency, Cleanup.

8.5.1 Agent's brain

The brain is what governs the body, not only in humans, in virtual agents too. It is a hub, where all perceived information are processed and according to them, a result is produced.

Providing structures that gather, process and provide output based on the information received from the environment [Picture 22]. Sensors gather information about the surroundings and events happening around the agent, based upon constriction e.g., range of vision, audible range, reach etc. The memory [Picture 10] provides a unified storing location for the gathered information. It can be accessed by the most important part of the agent's brain – the Action Selection Mechanism (ASM). The ASM can influence the stored memories and perform action by effectors (e.g., vocals) or agents extremities (hands). The agent has also a bottomless inventory at disposal, where it stores objects used to satisfy its goals.



Picture 22: Agent‘s brain

8.5.2 Path finding

Path finding is an important issue for agents. We use A* algorithm [46] to provide satisfactory results.

8.5.3 Be-tree

We implement the behavioral tree according to the specification, where the top level of top-level goals is anchored in the *action selection* and lower levels are linked. All nodes in the be-tree, except the actions in leafs, have to implement a common interface.

```
///executes the action
u_int execute();
///resets the execution
void reset();
///something has failed, a above plan
u_int fail();
///it is successfull
u_int success();
///the trigger of this execution has failed
u_int trigger_fail();
///the triggers of this execution rules should be checked
void trigger_check();
///switch this execution IN
u_int switch_in( const std::string& r);
///switch this eexecution OUT
u_int switch_out( const std::string& r);
///return the actual state of the execution
u_long_long state();
///is this execution interruptible or interrupt safe?
bool interrupt_safe();
///initialize an exit and exclude a set of object from the execution
u_int do_exit( std::list< u_long_long > exclude );
///collect what is left to cleanup when this has exited
void get_cleanup_residue(std::list< CollectedObject_id >& c_list );
///returns a set of exluding items from the cleanup process
void get_excludes( std::list< u_long_long >& c_list );
///needs an object
bool need_object( ObjectBase* o );
///get name
std::string get_name() { return std::string(); }
```

Every node in the be-tree is either instantiated or blueprints. When a blueprint node is accessed, it is first instantiated and then the execution proceeds.

8.5.4 Plan rules

A reactive plan consists mainly of rules, where a releaser, priority and action are the main body of a rule. Other proposed auxiliary data can be introduced into a rule. Triggers represent the releaser and actions are objects dedicated to either a sort of execution (can be specialized action e.g., fail, IaA) or forming a link to another reactive plan or similar structure (virtual nodes (7.6), extended actions (6.8)).

8.5.5 Actions

Actions are objects that maintain specific execution logic. We introduce a variety of actions

- External function excution
- Sequences & Cycles
- Specialized actions – success, fail, IaA
- Reactive plan links
- …

The action objects are derived from a common class *ActionExecution* implementing the action interface.

```
u_int execute();
```

We understand the action object as a container with unknown internal logic that can either return states of execution

- Fail – the execution has failed
- Success – successful execution
- None – the execution continues

Or throw specific exceptions (4.4.3)(7.6) that are caught either by nodes in the be-tree or the ASM.

8.5.6 Object/Class Location memory

The object/class location memory is a simple associative memory, where objects and classes are associated with their respective positions in the v-world. They can be recalled by action selection or by conditions in the rules releasers. When searches are performed, the information contained in the memory can be used as a hint where to go first.

A reinforcement and aging are introduced into the location memory, to provide more believable performance.

9 Summary

In [Chapter 2], we introduced the limitations of HRP concept. In this chapter, we summarize the proposed concepts to overcome the limitations. The problems of ASM observed are summarized in [Chapter 4] and the limitations of HRP are presented in [Chapter 5].

9.1 ASM related issues

- *Interrupting – behavior consistency* (4.1)

  To compensate for this drawback, we proposed usage of a specialized flag - *interrupt-safe flag* - not allowing action selection to surpass the active rule with a higher priority rule. The flag also suspends *releaser* checks and propagates up the SF-HRP hierarchy (a child sub-tree being *interrupt-safe* renders the parent *interrupt-safe*). By introducing the *interrupt-safe zones,* we intended to refine the approach to allow action sequences to be less non-responsive.

- *Releaser fault* (4.2)

  To compensate this problem of ASM, we propose usage of a *releaser-safe flag* that inhibits the releaser validation. A plan could also give a hint to the action selection, how to deal with a *releaser fault* if it happens – ignore it, switch or fail the befallen plan.

- *Delayed rule activation – sticky rules* (.3)

  To compensate for the problems caused by introducing *interrupt-safeness,* and later concepts (like switching), we propose to use a *sticky flag* with a *timeout*, to heed the demand for delaying rule activation, when their releaser holds, but they cannot be put to execution, even when supposed to. This flag doesn't propagate upwards in the hierarchy.

- *Fail and success* (4.4)
  - *Propagation of fail/success* (4.4.1)(4.4.3)

    To compensate for the limitation of simple fail/success propagation in HRP, we proposed to enrich the fail/success actions with a *counter* indicating how far the event propagates upward in the HRP's *be-tree* architecture. An *anti-counter* can be specified for given levels, introducing control to the spread of a fail/success event (anti-counter is subtracted from the counter, halting the spread when the counter reaches zero).
  - *Amount of fails/successes* (4.4.2)(4.4.4)

    Inspired by the POSH [36] we presented an approach to limit the amount of fails for a certain sub-tree, leading to disabling that branch. We propose to extend this approach to the success event, when a limited amount of successful executions is desired.

- (Un)biased Random Selection (4.5)

  We suggest expanding the rule formalism adopting a weight factor, providing a tool to specify preferences during random selection among equal priority preactive rules.

9.2 HRP related issues

Most of the issues presented could be tackled thanks to introducing the phases into the HRP model, extending it into the SF-HRP model. Also adding auxiliary structures (object sets, object class sets) and other concepts provided enough tools to overcome the presented limitations.

- *Intentions*
  - *Adding new intentions* (5.2.1)

    We propose to introduce a special set of actions – the *modifiers* (7.3) in conjunction with the IGPmap (7.1), providing specialized actions devoted to the adding of intentions to the *be-tree* – the IaA (7.6). The IaA can provide new top-level branch for the be-tree, based upon the entries in the IGPmap or explicitly constructed and stored in the IaA for the purpose of adding the specific setup of the new branch or the top-level of the *be-tree*.
  - *Choosing alternatives* (5.2.2)
    - *Process based view* (5.2.2.1)

      We drafted the concept of *goal-intention metadata* (6.3) to be included in the structure of a reactive plan, and by using it in tandem with extended action (6.8) – *goal based choice* – a suitable solution for this limitation can be employed.
    - *Object choice based view* (5.2.2.2)

      Introducing the *object sets* (6.4) and the *object class sets* (6.5) into the reactive plan structure, this constrain can be exceeded – the plan searches for objects it might make use of, choosing its alternatives based upon the environment (what is provided, how far it is, etc.)
- *Planning* (5.3)

  We propose the use of *object sets* (6.4) and *object class sets* (6.5) that allow the plans to better reason about needed resources. A specific phase was devoted to the task of acquiring of objects – initialization phase.
  - *(Cross plan) Preparation* (5.3.1)

    Peeking into other *preactive/suspended/sticky* plans object requirements can aid in mimicking perception of other tasks and their needs. The agent's action selection mechanism could analyze these requirements, providing the necessary reasoning, when and how to heed those needs (go and collect).

    By introducing the *focus* factor (5.31.1), the cross plan preparation can employed even for plans in different phases, not limited to the initialization phase of all the participants in collective item collecting.
  - *(Deep) preparation* (5.3.2)

    Parent plans can analyze object requirements of children plans (asking what they might require). Another possible approach is to explicitly specify the requirements of deeper situated plans. The concept of object requirements therefore can aid the process of mimicking "in forward" thinking.

    Keeping the sub plans from cleaning up their required objects can lead to more effective and believable behavior, by hindering the sub plans from reacquiring the same objects for every repetition of their execution.

    The parent plan can learn from the needs of children plans, by collecting their satisfied requirement after a successful execution.

- *Collecting objects (effectively)* (5.3.3)

  Thanks to the information contained in the *object set* and the *object class list*, the initialization phase can optimize searches and acquiring of objects (even for multiple plans in cross plan preparation).

- *Search for objects with (sharp) timeouts* (5.3.4)

  Introducing a simple timeout concept into the *object* and *object class sets* provides the ability to search with timeouts during the initialization phase. The in forward known requirements allow the initialization phase to overcome the issue of "seen object with an expired timeout". In specific situations, the timeout can be overruled. An object that might better suit the execution (it render a high importance list satisfied), even when its timeout has expired, providing more believable behavior in certain situations.

- *Plan chaining* (5.3.5)

  We drafted a plan chaining mechanism. Localization and object requirements of plans provide information that might influence the ASM's choice of candidates. The object requirement can be fetched from the plan object sets and the location stored per plan in an auxiliary map structure. The addressed bonus function should be designed per application, taking the world's properties into account.

- *Transitional behavior* (5.4)

  Transitional forms of behavior are natural to biological entities; therefore we introduced a system similar to the switching matrix proposed in [15]. The compensation for this limitation benefits from introducing *cleanup* and *switch* phases.

  - *Cleanup behavior* (5.5)

    The *cleanup* phase is dedicated to deal with the need to mimic *cleanup* behavior. Where a specialized structure devoted to performing the *cleanup phase*, can adopt various approaches for cleaning up objects. Also the possibility of excluding objects or omitting the phase entirely, can lead to a more believable behavior, spare unnecessary repetitive cleanups and acquisitions of objects, corruption of other plans execution context, etc.

  - *Switching of behaviors* (5.4)

    The trio of phases - *switch out* - *switched* – *switch in* handles the switching of behaviors. The proposed *default behavior* can be used to correct corrupted states, or crate a common switch point for all non-related behaviors. Where the *related behavior* is can be used to create behavior bridges between certain activities, rendering the overall execution appear more smooth.

    A proposed factor of *urgency* can provide a variety of switching behaviors, better adapting to the current situation.

- *Behavior aftermath* (5.6)

  The *termination* phase provides the means to employ behaviors dedicated to the outcome of the executed plan. Introducing floating-point results can provide an adaptable tool in creating better responses to agent's performance (in terms of fail and success).

9.3 Scenarios

In this chapter, we address the presented scenarios. There are 8 scenarios in total, containing various sub-scenarios, which illustrate various behaviors based upon SF-HRP.

9.3.1 Scenario 1 – "Hungry dog"

In scenario 1, we introduce the v-dog actor, who has a simple life – running around, feeding when food is in sight, barking occasionally.

The brain outline that mimics this behavior is presented in [Code 9].

```
a)  Priority [0], weight [20], releaser( true ), { bark, stop barking }
b)  Priority [0], weight [80], releaser( true ), { run around }
c)  Priority [1], weight [10], releaser( see(food) )
    (1) Priority [0], weight [100], releaser ( true ), { goto ( closest food )}
    (2) Priority [1], weight [100], releaser ( next_to_class(food) ) {eat, smile}
```

Code 9: v-Dog's brain

Without our improvements applied, the v-dogs brain won't behave in a believable manor. When food is presented in the middle of executing action sequence (a), v-dog won't stop barking, following the given impulse immediately. Therefore, we can make use of interrupt-safe (4.1) flag in rule (a). Other issue is the occasional barking and a lot of running around. Introducing weights (4.5) can provide the necessary preference specification for such a situation – the v-dog will run 80% of the time, barking at occasionally 20%.

Another issue at hand is the (c.2) rule, which corrupts the context of the (3) encapsulating plan, never allowing executing the "smile" action and failing as a consequence of the releaser corruption. Applying the *interrupt-safe* flag to rule (c.2), which propagates upwards, not allowing the releaser to fail. There is also a different solution, applying the use of a *releaser-safe* flag on the rule (c). The behavior of an always hungry dog can be observed in – sub-scenario 1.

The v-dog presented in (sub-scenario 1) eats every pieces of meat in sight. Where the v-dog presented in (sub-scenario 2) is modified, allowed only to eat 2 pieces of meat in one simulation run. To achieve this, we introduced the *propagated success* and provided the sub-plan specified in (c) with a finite amount of success equal to 2. The sub-plan under (c) will be disabled after the counter is used up.

9.3.2 Scenario 2 – "Sprinter dog"

```
(a) Priority [0], w[100], releaser( true ) { run (10 squares) } + interrupt-safe
(b) Priority [1], w[100], releaser( next-to(bone) ),{bark, stop barking}
(c) Priority [2], w[100], releaser( next-to(meat) ),{cheer}
```

Code 10: Runner v-Dog's brain

In Scenario 2 a sprinter v-dog is presented. His only thing to do is to run 10 meters (10 tiles) in the v-world (a). This sequence is interrupt-safe. V-dog is so fast, that it takes him 10 meters to slow down. During his sprint, he might see a v-bone or v-meat that might interest him (rule (b)(c)) in some way (barking, cheering etc.). On the other hand, when he runs too far, it might be less of interest to him, being "out of reach". To model this, a HRP

model could use complex conditions on one hand to let the v-dog finish his sprint, and later cheer for the seen bone/meat.

When employing *sticky rules*(4.3) with *timeouts*. Simply specifying that the dog should "keep the rule in mind" even when its releaser stopped to hold. This can be considered a primitive form of memory.

This scenario has 3 sub-scenarios. Different setups of sticky timeout are presented in [Table 10]

Rule in [Code 10]	Sub-scenario 1	Sub-scenario 2	Sub-scenario 3
(b) (bone)	1	12	12
(c) (meat)	12	1	12

Table 10: Sticky timeouts in Scenario 2

The v-dog will bark or cheer based upon the given timeouts, where the timeout "1" will expire at the next step, where the timeout "12" will present itself at the end of the sprint.

9.3.3 Scenario 3 – "The Collector"

In scenario 3, we show a simple agent, whose only goal is to get a set of objects in an optimal way (5.3.3), run around randomly and put the items back(5.5). The actual execution phase of the agents contains only a rule with the action for "random running". To simulate the need for the object by another preactive plan, the brain is equipped with a second plan with lower priority, but an always-holding releaser. The plans requirements on objects are based upon the sub-scenario. The Collector's brain is summarized in [Code 11], where (a) and (b) are plans on the top-level.

```
(a) Priority [0] rules {} plan with object requirements { hammer }

(b) Priority [1] rules { run around for N times },
                         object requirements { hammer, shovel, axe, watering-can }
```

Code 11: The Collector's brain

For (sub-scenario 5) the plan specified in (a) doesn't require any objects. Other sub-scenarios are specified in [Table 11]. The specified objects are required by the plan in (a) and therefore not cleaned up.

Sub-scenario	Object
1	v-axe
2	v-hammer
3	v-shovel
4	v-watering can
5	none

Table 11: Object requirement specification for Scenario 2

The optimal path to collect and bring them back will be chosen by the agents auxiliary structures, behaving in a more believable fashion, even when the objects change their positions or order. The object required by (a) in [Code 11] will be kept in the inventory.

9.3.4 Scenario 4 – "The Lumberjack"

In this scenario, we show an agent whose brain is a little more complex. His task is to cut down the trees in sight. That might not be that hard to do, but the axe gets blunt from all the work, so he might need a sharpening stone for it. The agent cheers after he is finished with his work and putting all things back from where he got them. Cheering is symbolized with the agent glowing for one cycle.

The structure of the agent's brain is shown in [Code 11].

```
Woodcutter plan: requirements {axe} terminate-success{glow} success limit{1}
                 Releaser { tree in sight }
                 after-initialization{ put axe into Right hand}
(a) see tree -> goto tree
(b) next to tree -> cut tree down
(c) blunt axe -> sharpen plan

Sharpen plan:  requirements {sharpening stone} terminate-success{glow}
               trigger-safe, after-initialization{ put stone into Left hand}
(a) true -> use Left hand on Right hand

                        Code 12: Lumberjack's brain schema
```

This is the basic outline of the Lumberjack's brain. The following sub-scenarios show how objects are acquired (5.3.2)(5.3.3), how the agent can express execution aftermath (5.6) behavior based upon the result and how deep plan preparation is employed. Also cleanup behavior is shown. The three provided sub-scenarios change the requirements of the plan in [Code 12].

Where in sub-scenario 1, the "Woodcutter" plan has only the axe, the "Sharpen" plan having the sharpening stone. When the sharpening is engaged, the stone is acquired, later when the sharpening is done, returned.

In sub-scenario 2, the "Sharpen" plan is marked as "no cleanup" hindering it from cleaning up the reused item. The item is later cleared by the "Woodcutter" plan.

In sub-scenario 3, the v-sharpening stone is made a requirement for the "Woodcutter" plan, acquiring it in forward along with the axe.

9.3.5 Scenario 5 – "The Saw"

In this scenario, we present a lumberjack employing a plan to cut trees using a saw. It is similar to the Woodcutter plan from (9.3.4) with the difference of a missing "sharpen" sub-plan [Code 12].

```
Sawcutter plan: requirements {saw|chainsaw} terminate-success{glow} success
limit{1}
                 Releaser { tree in sight }
                 after-initialization{ put into Right hand}
(a) see tree -> goto tree
(b) next to tree -> saw tree down

                        Code 12: Lumberjack's brain schema
```

Depending on the sub-scenario, the lumberjack is presented with various options for his *object requirements* for the "Sawcutter" plan – a v-saw or a v-chainsaw. The agent chooses the most optional way based upon the provided setup and the situation in the environment. In sub-scenarios 1 and 2 the objects are in various distances to the agent, the requirement specified by an *object class* (6.5) both objects share – "saw". The agent collects the closest of them and goes on with the plan. In sub-scenario 3 and 4, the object requirements are explicit by the unique identifiers, providing options either saw or chainsaw, or vice versa. The objects are equally distanced from the agent. In sub-scenario 5, both objects are specified explicitly in one object list, both are collected and one of them is used.

This scenario showed how *object base choices* (5.2.2.2) can be easily managed and are performed based upon the information published by the *object* and *object class sets.*

9.3.6 Scenario 6 – "Rich on options"

This scenario is the combination of (9.3.5) and (9.3.4), where the choice of an alternative is provided by a specialized action (6.8) – *random multi plan.* The setup of the agents mind is simple, shown in [Code 13].

```
Option plan: terminate-success{glow} success limit{1}
                Releaser { true }

Random Multi Plan     releaaser{true} counter[2 fails, 1 success]
          plan options{ Sawcutter plan, Woodcutter plan }
```

Code 13: Lumberjack's brain schema

The plan in [Code 13] is simple, containing only the choices for a "Sawcutter" and a "Woodcutter" plan. The choice is made random. This shows, how *process based alternatives* (5.2.2.1) can be chosen, either by random or any other way, depends on the specialized action employed. This scenario has to be run multiple times to see all the possible outcomes, all the sub-scenarios for scenario 5 and 4 can apply simultaneously.

The fail/success event counters are specified to provide two possible fails but only one success. The success satisfies the goal; therefore only one of the plans has to succeed.

A sub-scenario 6 is presented, where the *v-axe* is missing. A timeout (5.3.4) of 10 cycles is provided to show, how failed searches can result in an angry (glowing red) behavior.

9.3.7 Scenario 7 – "Warrior/Lumberjack"

In this scenario, we show how transitive (switch) behaviors (5.4) can be employed. The basic idea of the scenario is, that a warrior wakes up, gets his shield and axe and searches randomly for the forest to cut some trees down to build a fort[1]. When he finds some trees, he puts his shield on his back and goes on cutting some trees.

The outline of the warrior-agent's brain is [Code 14]

```
Warrior/Woodcutter brain:

Releaser(true) Priority[0] require{ axe, shield } -> walk random
Releaser( see tree ) Priority[1] -> Woodcutter
```

Code 14: Warrior/Lumberjack brain schema

[1] We addressed this scenario earlier in (5.4) to illustrate the same point.

The important about this scenario is the difference between sub-scenario 1 and sub-scenario 2. The first employing related transition behavior from “walking random” to “Woodcutter”, where the second one does only default switching behavior. The transitional behavior excludes the axe being put into the inventory of the agent. Where the default behavior puts both objects into the inventory and the “Woodcutter” plan takes the axe out, to be able to use it. After the warrior finishes felling trees, he continues on searching in equipping his shield and axe.

9.3.8 Scenario 8 – “Wandering Lumberjack”

This scenario is similar to scenario 7, where the agent wanders around and when he sees trees, he goes on and cuts them down. The important difference here is, that the agent’s brain look like this [Code 15].

```
Warrior/Woodcutter brain:

Releaser(true) Priority[0] -> walk random
Releaser( see tree ) Priority[1] {successes 1} -> add goal(woodcutter)


                    Code 15: Warrior/Lumberjack brain schema
```

Employing the IaA(7.6) concept, where a preconstructed plan is included in the IaA. As a result, the IaA action will append a goal to the *be-tree* structure, providing a plan “on-demand”(5.2.1). The resulting structure looks more familiar [Code 16]

```
Warrior/Woodcutter brain:

Releaser(true) Priority[0] -> walk random
Releaser( see tree ) Priority[1] {successes 1} -> add goal(woodcutter) //disabled
Releaser( see tree ) Priority[2] -> Woodcutter


                    Code 15: Warrior/Lumberjack brain schema
```

The plan itself after that, behaves very like a normal, designed version.

10 Conclusion

We provided a set of improvements and solutions that might aid to overcome the proposed limitations and delivering a more believable behavior to a human observer. However, the overall outcome depends of the v-world engine and the use of reactive plans concepts by the designers to create the agents mind. The improvements were overall dedicated to provide more control to the action selection mechanism and the plans themselves, to be able to „reason“ in a more suitable fashion, not only choosing and executing the rules from the *rule containers* – the reactive plans.

Proposed improvements consist of

- Interrupt-safe flag
- Releaser-safe flag
- Sticky flag
- Weight of rules
- Track flag
- Focus of rules and plans
- Propagating fail/success events
- Fail/Success counters for rules
- Object sets
- Object class sets
- Intention Goal metadata
- Phases for reactive plans – SF-HRP
    - Initialization phase
    - Termination phase
    - Switch IN/OUT phase
    - Switched phase
    - Finish phase
    - Cleanup phase
    - Execution phase
- Extended action selection mechanism
- Extended actions
- Be-tree modifiers and virtual nodes
- Intention add Action (IaA)
- (E)IGPmap
- Blueprints and instances

We think, we achieved a significant improvement to the concept of reactive planning, raising various new questions regarding the issues at hand. We tried to mimic the human perception of „how things are done“ to be able to provide the behavior in a more believable way, keeping the designer and the timely fashion of HRP in mind.

Some of the concepts are outlines, which might manifest themselves in a big scale highly dynamic simulation where complex minds are employed.

When employed, the designer has also to be taken into account, because a faulty methodic of creating minds based on SF-HRP with IaA and other concepts, can lead to a less believable behavior then with HRP.

We see a lot of potential in the phase-drive approach to reactive planning, because it maintains the core features of reactive planning providing auxiliary functionality that benefits the action selection mechanism to a considerable extent. The overall concept can be further developed, exploiting the modular approach of the resulting structure of SF-HRP.

We provided new entities to the *be-tree* structure, to provide online modification support to create more an adaptable concept as a result. Introducing various types of execution nodes providing a more versatile tool for the designer. The proposed auxiliary structures allow publishing necessary information about reactive plans, to be able to perform a simple but powerful analysis on the demands of the behaviors represented by those plans.

The SF-HRP model provides more control and information to the plan and the aside and above existing entities.

The motto of this thesis can be summarized as

„Give more control, get better behavior“

11 Future work

At this point, the SF-HRP concept lacks rigid testing in large-scale environments, employing complicated plans for agents. Providing an interesting subject to be implanted into a dynamic and complex simulation, to verify the characteristics on large scale. The ideas of SF-HRP can be individually deployed to provide agents with the ability to perform more believable behavior.

Per-plan memory provides an interesting field of further study, which could benefit not only SF-HRP, when fully deployed. Introducing emotions into the model can allow simulating human-like responses to situations. A hierarchical concept provides suitable ground to introduce planning into reactive planning in a larger extent.

Among future works, there is the further development of the online modifiable *be-tree,* following the ideas of modifier actions, channels and more advanced plan analysis. Other issues include parallelism and (E)IGPmaps. Introducing parallelism into SF-HRP can be considered the next step in the SF-HRP evolution.

Possible applications of this work might be in simulations with demand for complex believable behavior of autonomous agents, like computer games and virtual drama.

The concept being without a debugging or prototyping toolkit opens up a bunch of interesting themes.

Not to forget the ever-standing question of speed and reliability. In the actual draft of the SF-HRP concept is plenty of room for improvements in this field.

Bibliography

[1] EA Games (Maxis), URL: http://thesims2.ea.com/ (2004) [16.4.2009]

[2] Brom, C., Gemrot, J., Burkert, O., Kadlec, R., Bída, M. (2008): 3D Immersion in Virtual Agents Education, In Proceedings of First Joint International Conference on Interactive Digital Storytelling, Erfurt, Germany, Springer, Germany, p59-70

[3] Balicer, R. D. (2007): Modeling Infectious Diseases Dissemination Through Online Role-Playing Games, Epidemiology Volume 18 - Issue 2, p260-261

[4] Mateas, M., Stern, A. (2003): Facade: An Experiment in Building a Fully-Realized Interactive Drama, Game Developers Conference, Game Design track,

[5] Reidl, M. O., Stern, A.: Believable Agents and intelligent Scenario direction for Social and Cultural Leadership Training

[6] Bohemia Interactive: Virtual Battlespace 2, URL: http://virtualbattlespace.vbs2.com/ [16.4.2009]

[7] Predinger, H., Ishizuka, M.: Introducing the cast for social computing Life-like characters, In: Life-like characters. Tools, Affective Functions and Applications, Cognitive Technologies Series, Springer, Berlin

[8] Friedlander, D., S. Franklin, S.: LIDA and a Theory of Mind

[9] Blizzard Entertainment, URL: http://www.worldofwarcraft.com/index.xml [10.3.2009]

[10] Valve Software (2004) URL: http://orange.half-life2.com/ [12.4.2009]

[11] Epic Games (2007), URL: http://www.unrealtournament3.com/ [16.4.2009]

[12] Wooldridge, N.(2002): AN Introduction into MultiAgent Systems, John & Wiley & Sons

[13] project Agentfly, ČVUT, Fakulta Elektechnická, URL: http://agents.felk.cvut.cz/projects/agentfly/ [16.4.2009]

[14] Bryson, J. (2001): Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents. PhD thesis, Massachusetts Institute of Technology, 59-75.

[15] Mikula, T. (2006): Hierarchical reactive planning with transitions, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha (in Czech)

[16] Pfeifer, R., ScheierM C. (1999): Understanding Intelligence, MIT Press

[17] Brooks, R.A. (1986): A robust layered control system for a mobile robot, In: IEEE Journal of Robotics and Automation, RA-1, April, p14-23

[18] Lorenz, K. (1981): Foundations of Ethology, Springer-Verlag

[19] García, C.G., Geréz, P.P., Martínez, J.N.: Action Selection Properties in Software Simulated Agent

[20] Negatu, A.S., S.Franklin, S. (2000): An Action Selection Mechanism for „Conscious“ Software Agents, In: Cognitive Science Quarterly

[21] Bryson J., Stein L.A.: Modularity and Design in Reactive Intelligence

[22] Therau, C., Bauckhage, Sagerer, G. (2003): Combining self organizing maps nad multilayer perceptrons to learn Bot-Behavior for a commercial computer game, In: Proc. GAME-ON

[23] Brom, C., Gemrot, J., Bída, M., Burkert, O., Partington, S.J., Bryson J. (2006): POSH Tools for Game Agent Development by Students and Non-Programmers, In: CGAMES IEEE DUBLIN

[24] Brom, C.: Hierarchical reactive planning: Where is its limit?

[25] Champandard, A.J. (2003): AI Game Development: Synthetic Creatures with Learning and Reactive Behaviours, New Rider, Chapter 8

[26] Champandard, A.J. (2003): AI Game Development: Synthetic Creatures with Learning and Reactive Behaviours, New Rider, Chapter 3

[27] Kadlec, R. (2008): Evolution of intelligent agent behaviour in computer games, Master Thesis, Faculty of Mathematics and Physics, Charles University, Prague

[28] Champandard, A.J. (2003): AI Game Development: Synthetic Creatures with Learning and Reactive Behaviours, New Rider, Chapter 9

[29] Rabin, S. (2003): AI Game Programming Wisdom 2, Charles River Media, Inc, p303-317

[30] Rabin, S. (2003): AI Game Programming Wisdom 2, Charles River Media, Inc, p229-237

[31] Rabin, S. (2001): AI Game Programming Wisdom, Charles River Media, Inc (pathfinding)

[32] Isla, D.: Handling Complexity in the Halo2 AI URL: www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml [10.2.1009]

[34] Mateas, M., Stern A. (2004): A Behavior Language: Joint Action and Behavioral Idioms.

[35] Gill, S. (2004): Visual Finite State Machine AI Systems URL:

http://www.gamasutra.com/view/feature/2165/visual_finite_state_machine_ai_.php [16.3.2009]

[36] Chamapandard, A.J. (2007): Living with the Sims‘ AI: 21 Tricks to Adopt for your game, URL: http://aigamedev.com/reviews/the-sims-ai [16.4.2009]

[37] Forbus, D.F. (2002): Under the Hood of the Sims, CS 395 Game Design URL: http://www.cs.northwestern.edu/~forbus/c95-gd/lectures/The_Sims_Under_the_Hood_files/v3_document.htm

[38] URL: http://en.wikipedia.org/wiki/Pyrrhic_victory

[40] Blumberg, B.M. (1996): Old Tricks, New Dogs: Ethology and Interactive Creatures, PhD thesis, MIT Media Laboratory, Learning and Common Sense Selection

[41] Bryson, J. (2001): Intelligence by Design: Principles of Modularity and Coodination for Egineering Complex Adaptive Agents. PhD thesis, Massachusetts Institue of Technology,

[42] Tyrrel, T. (1993): Computational Mechanisms for Action Selection. Ph.D. Dissertation. Centre for Cognitive Science, University of Edinburgh, p185-187

[43] Gibson, J.J. (1979): The Ecological Approach to Visal Perception, Boston Houghton Miffin

[44] Bojar, O., Brom, C., Hladík, M., Vejlupek, M., Toman, V., Voňka, D. (2003): ENTs - A Simulator of Human-like Natural Environment. In: MIS proceedings, Matfyzpress, Czech Republic 3-14 (in Czech)

[45] Microsoft Game Studios (2004) URL: http://www.microsoft.com/Games/Halo2/ [16..4.2009]

[46] URL: http://en.wikipedia.org/wiki/A*_search_algorithm

Appendix A – Prototype

The prototype is available for Linux based systems only, possibly other POSIX systems too. Implemented in C++ language.

The basic requirements to run our prototype implementation are:

1) Linux based operating system with a kernel > 2.6.18
2) g++ or similar C++ compiler
3) SimpleDirectLayer (SDL) library (http://www.libsdl.org)

The prototype is simply installed by copying the provided installation directory "**prototype**" from the root of the CD-Rom into a new directory and executing

```
cd ./sources
make
cd ../bin
```

command from the console in the root directory containing the directory of the copied sources. A provided make script will create a binary in a separate directory denoted ***bin***.

To run the prototype, execute the created binary by issuing the command

```
./prototype
```

providing the following parameters denoting specific attributes. When used, they have to be followed by an integer number.

1) **T** (optional) amount of milliseconds between two cycles in the world
2) **S** (mandatory) the scenario to be presented
3) **U** (optional) a sub scenario, if available for that scenario
4) **C** (optional) number of maximum cycles to present
5) **J** (optional) jumps to the given step in the simulation

Example:

```
./prototype T200 S2 U1 C50 J10
```

- 200 milliseconds between cycles
- second scenario with the first sub scenario
- 50 cycles of the world
- jump to step 10

The application can be exited pressing the *escape* key and paused and unpaused using the *space* key.

The prototype provides detailed information on the execution, provided into the console window.

Appendix B – Related work

We derived most of the observed problems in this thesis from [15]. The paper discussed the issues of reactive planning, providing conceptual solutions to some of the problems. In conclusion some of the proposed concepts can be considered similar, but we intended to go a more structured way, introducing the SF-HRP.

The thesis [15] was concerned with transitional behavior, proposing a concept we build upon – the matrix of behavior pairs. Our related switching works with the same presumptions on the need for short pair identified behaviors. We extend this to the idea of IN/OUT behaviors and introduce a default switching behavior. The important difference is that our behavior switching can be considered safer, because it is limited to the execution phase, not invoked during pre- or post- execution phase. The SF-HRP structure provides a more controlled behavior.

The concept of reactive planning was derived from [21] introducing the concept. We derived and extended the proposed structures of basic reactive plans (BRP). Our work is oriented into virtual simulations and games, where J. Bryson's work is oriented more on the behavior of robots.

The ABL language [5] provides a similar approach of reactive planning to the issue of virtual narrative and modeling believable characters in social and cultural leadership training.

We also took inspiration from [35], describing how artificial intelligence of Halo 2 [45] computer game looks like. Implementing the AI exploiting the finite hierarchical finite state machines, providing a form of reactive planning to the non-player characters in the game. We try to propose a more broad based concept that can be employed with comparable easiness.

The importance of design toolkits should be omitted, therefore we consider the POSH [23] , Pogamut [2] and Ents [44] toolkits noteworthy, allowing to prototype agents employing reactive planning.

Appendix C – CD-ROM

The enclosed CD contains the sources of the prototype along with a PDF version of this text. The directory structure of the CD is described in a *readme.txt* in the root directory. The sources provided on the CD can be used freely, without any license restrictions.

The only request is to quote the author of this thesis when using any of his work in any way.